Software Approaches to Memory Latency Reduction for Scalable Shared-Memory Multiprocessors

Hui Li

Technical Report CSRI-328
June 1995

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Software Approaches to Memory Latency Reduction on Scalable Shared-Memory Multiprocessors

Hui Li

A dissertation submitted in conformity with the requirements
for the Degree of Doctor of Philosophy
Department of Computer Science
University of Toronto, January 1995

Abstract

Scalable shared-memory multiprocessors typically have distributed memory with various modules being relatively closer to some groups of processors than to others. From a particular processor, accesses to the farthest memory module can be a few orders of magnitude slower than accesses to the local one. The performance of these machines is thus constrained by average memory access latency.

Software techniques are needed for reducing memory latency. Without software support for attaining locality, data might be placed in a memory module far away from the processor using the data, significantly reducing performance. This thesis focuses on the development and evaluation of new software techniques that improve data locality in main memory and cache. These techniques are (1) a data transformation framework that optimizes data layout in shared-memory, (2) dynamic load balancing schemes that reduce mismatching accesses and achieve load balancing, and (3) a cache affinity-based scheme for cache coherence that reduces the frequency of coherence misses.

In uniprocessors, arrays are mapped onto memory in either row-major or column-major fashion. In scalable shared-memory multiprocessors, such mappings cause many unnecessary remote memory accesses. We propose data transformations that map the original arrays into desirable placements in the memory modules. Without changing the original array references in programs, the actual accesses to these arrays will be translated into less expressive accesses to mapped arrays. Complementary to loop transformations that mainly maximize parallelism, data transformations provide better data layout in the memory hierarchy, and improve data locality in both local memory and cache. We also develop a set of C macros, called NUMACROS, that implements data distributions (a subset of data transformations) and facilitates parallel programming with data distributions and loop partitioning.

Load balancing in parallel applications becomes an important issue when the iteration execution times or the task sizes have high variance. We develop a new loop scheduling technique, Locality-based Dynamic Scheduling (LDS), that allows for the variance in iteration execution times, and a new partitioning technique, called Overpartitioning, that allows for the variance of task sizes. LDS retains data locality while balancing load in scalable shared-memory environments where locality is important. Overpartitioning has been applied to different sorting algorithms, and the resulting algorithms outperform other parallel sorting methods on shared-memory machines.

The effective use of cache can significantly reduce average memory access time. Cache coherence is an important issue in multiprocessors. Most current compiler-directed schemes for cache coherence ignore cache reuse across parallel loops, resulting in many coherence misses. With increasing cache size, such cache reuse becomes more important to performance. We develop the Cache Affinity-based Scheme (CAS) which allows valid data to reside in caches across parallel loops, resulting in higher cache hit ratios than other software schemes that have been proposed.

We experimentally evaluate our new software techniques on two scalable multiprocessors Hector and the KSR1. Our benchmarks include a set of kernel parallel programs (such as matrix multiplication and LU decomposition), NAS Parallel benchmarks, and parallel sorting. With our software techniques, these kernels and applications achieve nearly linear speedups on both Hector and the KSR1.

Acknowledgements

Foremost, I would like to thank my supervisor, Ken Sevcik for introducing me to this area of research. His insight, guidance, and patience have been invaluable to me, and have been vital ingredients for the quality of this dissertation.

I would like to thank the other members of my committee, Tarek S. Abdelrahman, Christina Christara, G. Scott Graham, Mart Molle, and Songnian Zhou for providing valuable comments and suggestions during the various checkpoints required for the completion of this degree. Songnian Zhou always encouraged me when I was frustrated. Tarek S. Abdelrahman was always accessible for discussing anything from general ideas to machine-dependent details. Michael Stumm led a group that made Hector and Hurricane a reality, being used as an experimental platform for this research. I would also like to thank my external examiner, Tom LeBlanc for his careful reading of the dissertation and for his suggestions for improvements.

I would like to thank Sudarsan Tandri for many hours spent discussing various ideas and implementation approaches, as well as Dattatraya Kulkarni and Karim Harzallah for providing valuable feedback on early drafts of this work. I also wish to thank many people in the Parallel Discussion Group and the Compiler Discussion Group for their suggestions that improved my presentations.

Finally, I thank my wife for her love and sacrifices and my parents for their moral support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis focuses on reduction of memory latency for scalable shared-memory multiprocessors. In this chapter, we describe types of multiprocessors, define different types of remote memory accesses and cache misses, discuss related work, and preview the contributions of this thesis.

1.1 Types of Multiprocessors

Multiprocessing is the only plausible way to increase the computational power available to scientific and commercial applications. Many multiprocessors have been developed since the late 1980's with a variety of architectures. These multiprocessors can be classified by their memory organization. The main hardware distinction is whether the system maintains a single address space and a single coherent memory.

Message-Passing Multiprocessors. A simple way to build multiprocessors is to connect independent processors, each with its own physical memory, by a fast interconnection network. Such machines support multiple address spaces and explicit messages are required to access memory on other processors. These machines are thus called *message-passing* multiprocessors. Examples of such multiprocessors are CM5, nCube, Intel Paragon, and IBM SP2 [Bel92]. On such multiprocessors, programmers must deal with separate address spaces, synchronizing processors, and communicating data using messages. This is time-consuming and error-prone.

Small-Scale Shared-Memory Multiprocessors. A natural extension of uniprocessors is *shared-memory* multiprocessors that provide a single address space to programmers. Shared-memory multiprocessors based on buses have been commercially successful because they offer a better programming model and better support for standard operating systems and programming environments than message-passing multiprocessors. Examples of such multiprocessors are Sequent Symmetry, Encore, Alliant, and Convex, and more recently SGI Challenge and SUN SPARC [Bel92]. Since the access time to any memory location from any processor is the same, such multiprocessors are usually called *Uniform Memory Access (UMA)* multiprocessors. However, they do not scale well because of the limitation of bus bandwidth.

Scalable Shared-Memory Multiprocessors. To build a scalable multiprocessor, the memory system must be organized in a distributed fashion [Bel92]. Typically, each memory module is local to some processor, and all processors can access all memory modules through an interconnection network in such a way that the shared-memory paradigm remains. Since accesses to remote memory take longer than accesses to local memory, such multiprocessors are usually called *Non-Uniform Memory Access (NUMA)* machines.

Caches are widely used for reducing memory access latency. Multiple copies of data in different processors' caches introduce the problem of cache coherence: If any processor updates the data, other processors should be able to access the most recent value. Many multiprocessors provide hardware support for cache coherence, and they are called *Cache-Coherent NUMA* (or *CC-NUMA*) multiprocessors. Examples of such multiprocessors are the Stanford Dash and Flash [LLG+92], and the MIT Alewife [ACJ+91]. Other multiprocessors rely on software to enforce cache coherence. Examples of such multiprocessors are the BBN TC 2000 [BBN90], IBM RP3 [Bel92], Myrias [BBM+89], Cedar [KTV+91], Hector [VSWL91], and the Cray T3D [CRI92].

With increasing cache sizes on recent multiprocessors, the distinction between UMA and NUMA is diminishing. Current RISC processors typically consist of a few megabytes of secondary cache (*e.g.*, 4 MB on the MIPS R4400). Moreover, the next generation of processors will have ten times larger cache (*e.g.*, 96 MB on-chip cache on the Alpha 21164) [Gep95]. Therefore, large caches on small-scale shared-memory multiprocessors bring in a new level of memory access times, causing recent small-scale shared-memory machines such as SGI Challenge to resemble NUMA machines.

Cache Only Memory Architectures. In a *Cache Only Memory Architecture (COMA)*, the memory organization is similar to that of a NUMA machine where each processor has a memory module. However, these memory modules resemble a large (second level) cache. An address becomes a name and this name automatically migrates and replicates throughout the system and is associated with a processor in a cache-like fashion as needed. The Kendall Square Research KSR1 and KSR2 [KSR91] are examples of COMA multiprocessors.

Remote memory access times on scalable shared-memory multiprocessors vary from a few times slower to a few orders of magnitude than local ones. With an increasing number of processors, the range of access times to different memory levels becomes even wider. Fortunately, in most scientific applications, the working sets are much smaller than the data sets, and grow slowly as the problem size and the number of processors increase [RSG93]. Therefore, the working set of each processor can usually fit into a low level of the memory hierarchy as seen from that processor.

1.2 Problem Identification

Despite the shared-memory paradigm supported by hardware in scalable shared-memory multiprocessors, parallel programs designed for UMA machines often cause many unnecessary remote memory accesses, thus delivering little speedup or even a slow-down. This situation forces programmers to restructure their codes to improve performance, usually by improving the locality of references by each processor. Restructuring is a tedious, error-prone, and time-consuming task that requires detailed knowledge of the architecture and operating system. The restructured programs may achieve good performance, but they are hard to understand and quite machine dependent. Moreover, the programmers feel that programming on such shared-memory machines is as difficult as, or even more difficult than, programming on message-passing machines. The frustration of parallel programming on scalable machines is due to the lack of software support for reducing the number of remote memory accesses.

In this section, we classify remote memory accesses and cache misses, and then discuss current software techniques for reducing memory access latency.

1.2.1 Memory Latency

Remote memory accesses in a parallel application can be classified into the following four categories:

- *Communication accesses.* Any non-trivial parallel program requires some communications among processors. These communications are realized by explicit messages on message-passing multiprocessors, and by remote memory accesses on shared-memory multiprocessors. They are inherent to the parallel algorithms.

- *Synchronization accesses.* Since processors run asynchronously, synchronization is required to guarantee correct program execution. On shared-memory machines, synchronization is implemented by atomic memory accesses. These accesses are often remote.

- *Mismatching accesses.* When data in one processor's memory module is needed for computation on another processor, a mismatch occurs. When the execution times of the iterations of a parallel loop have high variability, the dynamic assignment of computation is required to achieve load balancing. However, dynamic load balancing schemes can cause additional mismatching remote accesses.

- *False-sharing accesses.* Due to large page size, two or more processors may access the same page without sharing data. This is called false sharing. No matter in which local memory the page is placed, such false sharing causes some remote memory accesses.

A cache holds the most recently referenced data to satisfy subsequent memory requests with minimal cost. Cache misses occur when the data to be accessed is not found in the cache. In uniprocessors, there are three types of cache misses: compulsory, capacity, and conflict misses [HP90]. In scalable shared-memory multiprocessors, cache coherence schemes may cause additional misses because, to guarantee the program correctness, they are usually conservative. These cache misses are called *coherence misses*.

1.2.2 Memory Latency Reduction

Software support for memory latency reduction is the key to the success of scalable shared-memory multiprocessors. Current software techniques for reducing memory latency include (1) loop transformations that reduce the working sets of applications to fit into processor caches, and (2) data management methods that place data in the local memory of the processor that will use it most.

Loop Transformations

Loop transformations were originally developed for improving loop parallelism [Wol89b]. Wolfe shows how tiling (or iteration space blocking) can be used to improve the memory performance of program loops [Wol87] and later presents algorithms that derive loop transformations from dependence distance vectors for tiling [Wol89a]. Irigoin and Triolet propose a dependence cone to block code for two levels of parallelism and two levels of memory [IT88]. Gannon *et al.* present a technique to describe the amount of data that must be in the cache for reuse to be possible [GJG88]. They call this the *reference window*. A family of reference windows for an array represent all of its elements that must fit into the cache to capture all of the reuse. If the cache is too small, blocking transformations can be used to decrease the size of the reference windows. Porterfield proposes a method to determine when the data referenced in a loop does not fit entirely in cache [Por89]. He defines the concept of an overflow iteration, which is the iteration of a loop that brings in data that causes the cache size to be exceeded. Using this concept, it is possible to predict when a loop nest needs to be tiled. He also discusses the applicability of blocking transformations. Wolf and Lam present a framework for determining memory usage within loop nests and use that framework to derive a unimodular loop transformation, tiling, and unroll-and-jam [WL91, Wol92]. Kennedy and McKinley present a simplified model of cache performance that only considers inner-loop reuse [KM92]. Using this model, they describe a method to determine the number of cache lines required by a loop if it were innermost and then they reorder the loops to use the minimum number of cache lines. Carr proposes algorithms that can block complicated iteration spaces, such as trapezoidal, rhomboidal, and triangular, and evaluates these algorithms on the IBM RS6000 [Car93].

Most loop transformations minimize cache capacity misses by reducing working set size, but they do not deal with cache conflict misses that depend on data layout in memory and cache. Lam, Rothberg, and Wolf show that typical effective block sizes use less than 10% of the cache to avoid cache conflicts for achieving a high cache hit ratio [LRW91]. They suggest the use of copy optimization to remove such effects and to allow the use of a larger portion of the cache.

Data Management

Data management includes techniques such as annotation of data distributions among processors by programmers, incorporation of new page placement policies in the operating systems, and maintenance of cache coherence in software by the operating system or compilers.

Data Distributions. Data distribution directives have been proposed to support data parallel programming for message-passing multiprocessors. Programmers can still use the shared-memory paradigm, but apply data distribution annotations to obtain desirable data decompositions. The research group led by Kennedy has developed High Performance Fortran (HPF) [HPF93], a successor to Fortran D [FHK+90], for portable parallel programming. The main feature in HPF is a set of data distribution directives. They propose the *owner–computes* rule to generate message-passing code from data distributions for message-passing multiprocessors and develop various compiler optimization techniques for improving communication efficiency [FHK+90, HKT91, HKT92]. Similar efforts include Kali [KMV90, KM91], C* [RS87], Dataparallel C [QH91, QHS91, HLJ+91, HQA+91], and DINO [RSW90]. Data distributions are also useful for improving data locality on shared-memory multiprocessors, but require different optimization techniques. For example, the owner-computes rule simplifies the generation of message-passing code from data distributions for message-passing multiprocessors, but it is not needed for shared-memory multiprocessors.

Page Placement. Research in the operating systems community focuses on simple and universal memory management policies to exploit data locality. This is motivated by the success of algorithms that approximate LRU for virtual memory page replacement on uniprocessors [PS85]. Black *et al.* propose competitive algorithms for page migration and replication, but they only present their results as compared against a random placement of data [BGW89]. Bollosky *et al.* conduct a trace-based simulation study of the performance of several migration and replication policies [BFS89, BSF+91]. Cox and Flowler's PLATINUM implements a small number of well-tuned policies in the kernel of the BBN Butterfly [CF89]. LaRowe *et al.* compare a set of page placement policies by implementing many alternatives in their DUnX version of the operating system for the Butterfly [LWE91, LE90, LEK91]. They conclude that no universal policies have been found capable of optimizing data locality for a large set of parallel applications on multiprocessors. Thus, application specific policies must be implemented outside the operating systems.

Distributed Shared-Memory. For scalable shared-memory multiprocessors without hardware cache coherence, several systems use distributed shared-memory protocols to implement cache coherence at a page granularity in operating systems [Li86, CBZ91, ZSLW92]. IVY is the first distributed shared-memory system that provides a shared-memory model on a network of workstations. While directory-based hardware schemes activate cache coherence operations by cache misses, distributed shared-memory systems rely on virtual memory page faults to trigger the cache coherence operations. However, the large page size causes extensive false sharing in many applications, and the high page-fault handling cost limits performance. To overcome these problems, a few recent systems, such as Munin [CBZ91], provide several types of coherence operations for different access patterns. But they require substantial effort from programmers to classify data access patterns and to insert cache operations in programs. The shared regions approach [SGZ93] provides a high-level abstraction making it easy for programmers to maintain cache coherence. However, it still requires programmers to identify which memory references require a remote read/write, which is nearly as tedious as inserting send/receive primitives for message-passing systems.

Compiler-directed Cache Coherence. Compiler-directed software schemes maintain cache coherence at a fine granularity. They utilize program analysis information to insert cache operations for parallel loops at compile-time. These cache operations are cheaper than kernel-level page fault handling. Fast Selective Invalidation (FSI) [CV88a, CV88b] invalidates the entire cache before each parallel loop, but cache reuse within a single parallel loop is captured by an extra bit added in each cache line. Software Cache Coherence through Vectorization (SCTV) uses vectorization techniques to eliminate redundant invalidations [DMCK92]. Both of these schemes, however, only consider single parallel loops, so the cached data cannot be reused across parallel loops. Without exploiting such cache affinity, unnecessarily high miss ratios may result. The Version Control [CV90] and Time Stamp [MB89, MB90] schemes can retain cache reuse in adjacent parallel loops, but they require a few additional bits to be added into each cache line to maintain the version number. In these schemes, a reference is treated as a cache hit if the version number is not less than it should be, otherwise the cache copy is stale and the updated value must be loaded from memory. However, the cost for maintaining the version numbers is high. The Life Span strategy has been proposed to reduce such cost by adding one more bit per cache line than FSI, but it exploits cache affinity between only two consecutive parallel loops [Che92].

1.3 Experimental Platforms

Two shared-memory multiprocessors are used in our experimental studies: Hector and the KSR1. We describe the architectures of these two machines.

1.3.1 The Hector Multiprocessor

Hector is a NUMA multiprocessor with a hierarchical structure [VSWL91]. Each processor module consists of a 16.67 MHz Motorola MC88100 CPU, a 16 KB instruction cache, a 16 KB data cache, and 4 MB of globally addressable memory. In the Hector hierarchy, the lowest level is called a *station*. A station contains

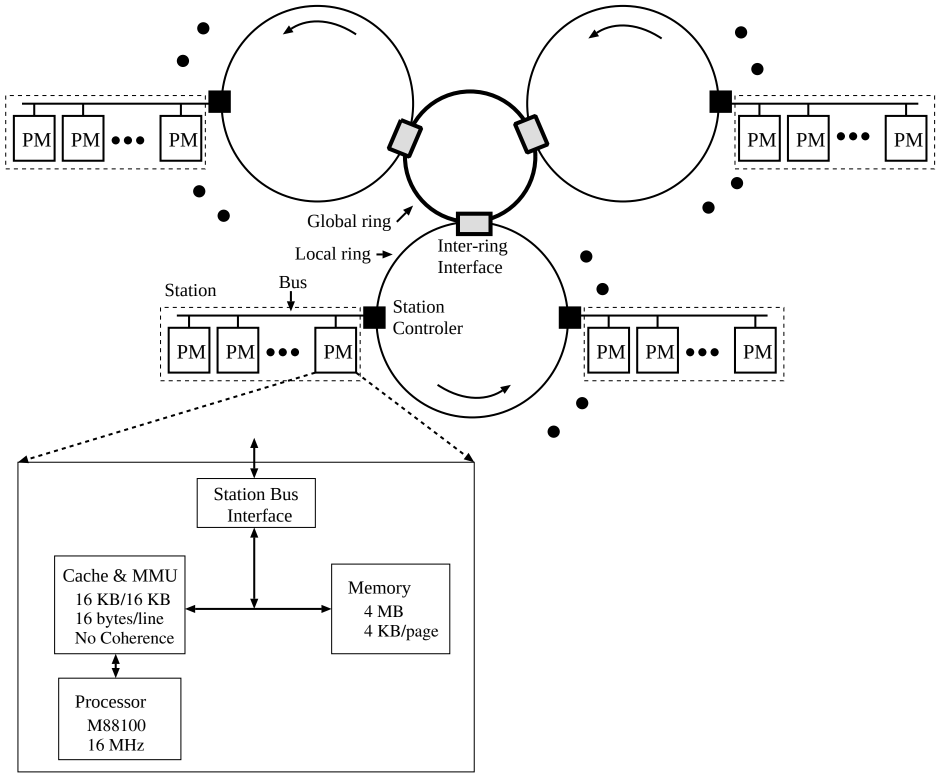


Figure 1.1: The Hector Multiprocessor

a number of processor modules connected by a bus. The second level is called a *local ring*. A local ring contains a number of stations connected by a bit-parallel slotted ring. At the top level, local rings can be connected by a *global ring* (or even a hierarchy of rings) to support a large system (Figure 1.1). The prototype of Hector used for the experiments in this thesis consists of 4 stations, each containing 4 processor modules, for a total of 16 processors.

Hurricane is a multithreaded operating system for Hector that supports process creation, process scheduling, and page-based virtual memory management. The page placement policies in the Hurricane memory manager include [Unr93]

- Round-Robin: pages are assigned to processor modules in round-robin fashion, thereby avoiding hotspots, but ignoring data locality.
- First-Hit: each page is placed in the memory of the processor that first touches it, in an attempt to achieve data locality. In many parallel applications, however, one processor initializes most variables, leading to very poor data placement under this policy. But we can use this policy to assign pages by causing each processor to first touch the pages assigned to its local memory.

Since there is no hardware support for cache coherence in Hector, the Hurricane memory manager maintains cache coherence at a 4 KB page level of granularity by permitting only unshared and read-shared pages to be cached [Unr93].

1.3.2 The KSR1 Multiprocessor

The Kendall Square Research KSR1, a COMA multiprocessor, is also configured as a hierarchy of slotted, packetized rings with processing cells on the leaf-level rings (see Figure 1.2). Each leaf level ring, ring:0, consists of 32 processing cells and two cells for routing to the higher level ring - ring:1. The KSR1 pioneers the *ALLCACHE* system where the local memory is managed as a second level cache. The ALLCACHE

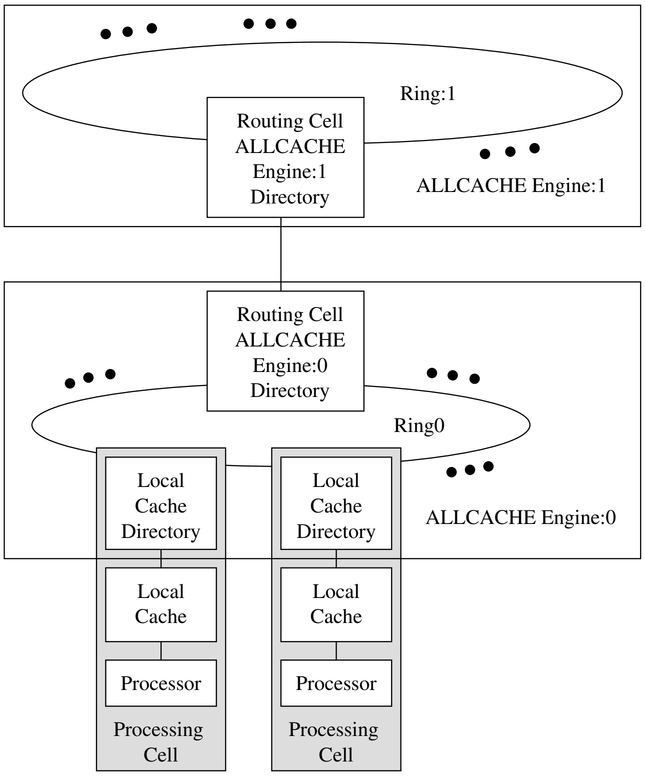


Figure 1.2: The KSR1 Multiprocessor

engine is responsible for migrating data to and from other processor cells, for maintaining memory coherence throughout the system using distributed directories, and ring control. Each processing cell contains a 64-bit superscalar processor at the 20 MHz clock-rate, a 256 KB instruction subcache, a 256 KB data subcache, and a 32 MB cache memory. The line size in the data subcache is 64 bytes, and the unit of transfer on the ring is one subpage (128 bytes). Each remote access request travels around the ring, so a snooping protocol can be used for locating a subpage within a single ring. When it reaches the ALLCACHE directory, if the lookup determines that a request cannot be satisfied in the leaf level ring, the request is transferred to the ring:1. The 64-processor KSR1 on two rings at University of Washington and the 32-processor KSR1 on a single ring at University of Toronto were used in the experiments for this research.

Table 1.1 gives the memory access times to different levels on Hector and the KSR1. Since remote memory accesses are much more expensive than local memory accesses, the number of unnecessary remote memory accesses due to mismatching and false sharing must be minimized by placing data on the processor that uses it most.

		Cache	Local	Remote-1	Remote-2
	Data Size (Bytes)	4	16	16	16
Hector	Processors			4	16
	Latency (cycles)	1	10	19	29
	Data Size (Bytes)	8	64	128	128
KSR1	Processors			32	64
	Latency (cycles)	2	18	170	530

Table 1.1: Memory Access Latencies

	Remote Memory Accesses	Cache Hit Ratio	Load Balance	Cache Coherence
NUMACROS	X			X
Data Transformations	X	X		
LDS	X		X	
Overpartitioning	X		X	
CAS		X		X

Table 1.2: Contributions of the Thesis

Hector is a NUMA multiprocessor without hardware cache coherence, so both data placement and the manner of guaranteeing cache coherence in software are important to performance. The KSR1 is a COMA multiprocessor that supports data replication and migration by hardware. Therefore, initial data placement has less performance impact. However, data reuse must be exploited in order to minimize the number of expensive remote accesses.

1.4 Preview of Contributions

This thesis focuses on the development and evaluation of new software techniques that avoid unnecessary remote memory accesses and unnecessary cache misses. We assume that parallelism is identified by parallelizing compilers or by programmer annotations. Our goal is to facilitate parallel programming and to achieve high performance on scalable shared-memory multiprocessors. The main contributions of this thesis are several software techniques that reduce mismatching and false sharing remote accesses and coherence cache misses in parallel scientific applications. These techniques are (1) a data transformation framework that optimizes data layout in shared-memory and improves cache locality, (2) dynamic load balancing schemes that reduce remote accesses and achieve load balancing, and (3) a cache affinity-based scheme for cache coherence that reduces coherence misses (see Table 1.2). These techniques can be incorporated into parallelizing compilers.

- **Data Transformations.** In uniprocessors, arrays are mapped onto memory in either row-major or column-major fashion. In scalable shared-memory multiprocessors, such mappings cause many unnecessary remote memory accesses. We propose data transformations that map original arrays into desirable forms in the memory modules [LS93]. Examples of data transformations are array transposition, array skewing, and regular data distributions from HPF (such as block, cyclic, and block-cyclic). Without changing original array references in programs, the actual accesses to these arrays will be translated into less expensive accesses to mapped arrays. Orthogonal to loop transformations that mainly maximize parallelism, data transformations provide better data layout in the memory hierarchy, and improve data locality in both local memory and cache. Chapter 2 will present a set of C macros, called NUMACROS, that implements data distributions (a subset of data transformations) and facilitates parallel programming with data distributions and loop partitioning. The general framework of data transformations will be described in Chapter 3.

- **Load Balancing.** Load balancing in parallel applications becomes an important issue when iteration execution times or task sizes have high variance. Dynamic loop scheduling schemes have been proposed to deal with the high variance in iteration execution times, but they do not take data locality into account, resulting in excessive mismatching remote accesses on scalable multiprocessors. We develop Locality-based Dynamic loop Scheduling (LDS) that uses data distribution information to retain data locality and applies global dynamic loop scheduling schemes to maintain load balance [LTSS93]. Some applications are inherently difficult to partition into equal sized tasks. An example of such applications is the parallel sorting problem. Current techniques for these problems attempt to retain data locality by making the number of partitions equal to the number of processors. However, this makes load balancing difficult. We propose the Overpartitioning technique and apply it to parallel sorting [LS94b].

It achieves good load balance as well as data locality. LDS and the Overpartitioning technique will be presented in Chapter 4.

- **Cache Affinity-based Software Cache Coherence.** The effective use of cache can significantly reduce average memory access time, so cache coherence is an important issue in multiprocessors. Most current compiler-directed schemes for cache coherence ignore cache reuse across parallel loops, resulting in many coherence misses. With increasing cache size, such cache reuse becomes more important to performance. We develop the Cache Affinity-based Software cache coherence scheme (CAS), which allows valid data to reside in caches across parallel loops, resulting in higher cache hit ratios [LS94a]. CAS will be described in Chapter 5.

We experimentally evaluate our new software techniques on two scalable multiprocessors, Hector and the KSR1. Our benchmarks include a set of kernel parallel programs (such as matrix multiplication and LU decomposition), NAS Parallel benchmarks, and parallel sorting. With our software techniques, they achieve nearly linear speedups on both Hector and the KSR1.

The proposed software techniques facilitate parallel programming in scalable shared-memory multiprocessors:

- NUMACROS allows programmers to annotate data distributions to exploit data locality without worrying about the details of operating systems and architectures. Parallel programs in NUMACROS are nearly as readable as sequential ones, and nearly as efficient as fine-tuned ones.

- By applying data transformations, arrays are transformed into desired forms to further improve cache locality and memory locality. Yet, programmers may still view the arrays in the original forms that are natural to represent their problems.

- The new cache coherence schemes provide coherent memory to programmers to relieve them from tedious cache operations. This makes programming on NUMA multiprocessors without hardware coherence as easy as on cache-coherent NUMA multiprocessors.

- For applications with load imbalance, LDS and overpartitioning techniques not only achieve good load balancing, but also relieve programmers from the details of determining subtasks and scheduling them onto processors.

The NUMACROS, data transformations, and CAS can be implemented in parallelizing compilers. The LDS and overpartitioning techniques for load balancing can be incorporated into run-time libraries.

Chapter 2

Parallel Programming in NUMACROS

In scalable shared-memory multiprocessors, shared-memory is supported by hardware for ease of parallel programming. Parallel programs designed for UMA multiprocessors can thus be executed on scalable shared-memory multiprocessors without modifications. However, if the need for data locality is ignored, such programs give little speedup or even a slow-down. Fine-tuning parallel programs for scalable shared-memory multiprocessors is a tedious, error-prone, and time-consuming task that requires detailed knowledge of the architecture and operating system.

To facilitate parallel programming for scalable shared-memory multiprocessors, we should provide high level notations for data locality. Data distributions have been proposed in many data parallel programming languages, including C* [RS87], Dataparallel C [QH91, QHS91, HLJ+91, HQA+91], Kali [KMV90, KM91], DINO [RSW90], Fortran D [FHK+90], High Performance Fortran (HPF) [HPF93], Superb [ZBG86], and NESL [BCH+93]. These languages are primarily designed for message-passing machines and their compilers are usually based on the owner-computes rule to generate message-passing primitives. However, parallel programming on scalable shared-memory ultiprocessors differs from that on message-passing machines in the following ways:

- Communication between processors using remote memory accesses is far more efficient than message-passing.
- Data is stored in the same address space, and therefore the owner-computes rule is not necessary.
- Data layout in main memory has great impact on data locality, false sharing, and cache hit ratio.

Therefore, we propose a set of macros, called NUMACROS, that facilitates parallel programming for scalable shared-memory multiprocessors. NUMACROS includes various data distributions and parallel loop constructs for achieving data locality and parallelism. We show that parallel programs written in NUMACROS can be as readable and simple as sequential programs.

In this chapter, we begin with an example using a parallel LU decomposition to illustrate the challenge of parallel programming and performance issues. We then show how NUMACROS can be efficiently implemented on scalable shared-memory multiprocessors and present the results of experiments on Hector. The effective, yet simple, implementation of NUMACROS demonstrates that scalable shared-memory multiprocessors provide better support for data parallel programming than message-passing machines.

For COMA multiprocessors, data distributions are not necessary since data replication and migration are supported by hardware. However, the data layout in memory may impact cache locality. We will generalize data distributions for improving both memory and cache locality in the next chapter, and we leave further treatment of other issues (load balancing and cache coherence) to Chapters 4 and 5.

```
double A[N][N];                              double A[N][N];
main()                                       main()
{  int i, j, k;                              {  int i, j, k;
   ... ...                                      ... ...
   for (k = 0; k<N-1; k++)                      for (k = 0; k<N-1; k++)
     for (j= k+1; j<N; j++) {                     do_cyc(j=k+1; j<N; j++)
       A[j][k] /= A[k][k];                           A[j][k] /= A[k][k];
       for ( i = k+1; i<N; i++)                     for (i = k+1; i<N; i++)
           A[j][i] -= A[j][k]*A[k][i];                A[j][i] -= A[j][k]*A[k][i];
     }                                            enddo
}                                            }
     (a) sequential version                       (b) parallel version
```

Figure 2.1: Parallel LU Decomposition for UMA Multiprocessors

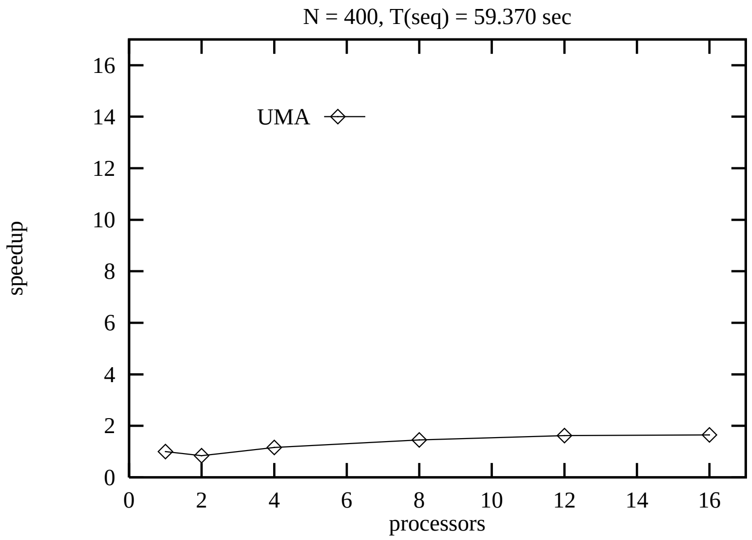


Figure 2.2: Speedup of the UMA Version of LU Decomposition on Hector

2.1 Parallel LU Decomposition

Figure 2.1 (a) shows a program for a sequential LU decomposition, which factors a dense matrix into an upper triangular matrix and a lower triangular matrix. It consists of three nested loops, where the middle loop, on index j, is parallelized. In a *parallel loop*, iterations are independent of one another without data dependences. For shared-memory machines, the parallel version (in Figure 2.1 (b)) replaces the sequential loop j by a parallel loop construct, `do_cyc`, that distributes the iterations evenly among threads (or processors) [1] in a round-robin fashion. The simplicity of the parallel version is due to the shared-memory paradigm. A version of this program for message-passing multiprocessors is much longer and more difficult to specify and understand.

This parallel LU decomposition can achieve decent speedups on UMA shared-memory machines. Without modification, it can also run on scalable shared-memory machines such as Hector. However, the speedup

[1] The number of threads we create equals the number of processors being used, so that the context switching overhead is minimized.

(shown in Figure 2.2) is embarrassingly poor. This is because most memory references in this parallel version become remote on Hector. Based on the classification introduced in Section 1.2, remote memory accesses in this program are:

- Communication accesses. In the k^{th} iteration of the outermost loop, every processor uses the pivoting row that was produced in the $(k-1)^{th}$ iteration. The current pivoting row is remote to all processors except one. The ratio of communication to computation is small, with $O(N-k)$ remote accesses and $O((N-k)^2/P)$ arithmetic operations per processor, where P is the total number of processors being used for the execution.

- Synchronization accesses. A barrier is associated with `enddo` to ensure that every processor finishes the current loop before any one can start the next one. Since there is only one barrier per parallel loop, this cost is negligible.

- Mismatching accesses. The default page placement policy on Hector is Round-Robin, so the pages for array A are allocated to the local memories of all the processors in the system (P_{all} of them) as follows: page i is assigned to processor ($i \bmod P_{all}$). Although `do_cyc` schedules iterations among the P threads in a round-robin fashion as well, a mismatch between page placement and thread scheduling occurs when $P \neq P_{all}$, thread i is not scheduled on processor ($i \bmod P_{all}$). Such mismatching causes most memory accesses to be remote.

- False-sharing accesses. If the row size is not a multiple of the page size, some pages contain different rows that are frequently accessed by different processors without actually sharing data, resulting in false sharing.

Besides remote accesses, this program also suffers from excessive cache misses. Since array A is read-write/write-write shared among all processors, the pages for array A become uncacheable under the Hurricane page-based cache coherence scheme, resulting in an even higher average memory access time. A better cache coherence scheme can retain cache line spatial locality, resulting in cache hit ratio at least 0.5 for the LU decomposition on Hector (with a cache line size of 16 bytes).

There are several simple ways to reduce the number of mismatching and false sharing remote memory accesses and to improve cache coherence.

- **Choosing Appropriate Policies.** To obtain better locality, we can cause row j to be touched initially by processor ($j \bmod P$) under the First-Hit page placement policy. However, false sharing still remains whenever the row size is not a multiple of the page size.

- **Enlarging the Array Size.** False sharing is caused by the rows not aligning with the page boundaries; we can simply enlarge the row size of array A to be a multiple of the page size. However, this not only wastes memory space, but also increases cache misses substantially because a multiple-page row size causes many rows to be mapped onto the same cache lines [LRW91]. In addition, it cannot be generalized to handle multi-dimensional arrays.

- **Maintaining Cache Coherence at the User Level.** Read-write/write-write shared pages can cause invalidation thrashing if they are allowed to be cached as in Distributed Shared-Memory (DSM) systems [Li86]. To avoid this problem, Hurricane permits only unshared or read-shared pages to be cached. However, it is too conservative to make most pages uncacheable. Compiler-directed schemes maintain coherence at a fine granularity. Within parallel loops, there are no data dependences, thus no read-write/write-write sharing. Based on this observation, several existing compiler-directed schemes [CV88b, CV88a, Che92, CKM88, DMCK92] have been proposed to maintain cache coherence for individual parallel loops. An example of these schemes is Fast Selective Invalidation (FSI) [CV88a], which makes all data cacheable within parallel loops by invalidating the entire cache before entering parallel loops. We will propose a new cache coherence scheme that exploits cache affinity across parallel loops in Chapter 5. In Chapters 2 through 4, we assume that cache coherence is maintained by FSI on Hector.

`dist_1D(A, 1, distType, i, sizeI, blkSize)`

- `dist_1D` distributes array `A` along its first dimension on the 1-D grid of processors.
- `distType` can be `block, cyclic`, and `block_cyclic`.
- `i` is a dummy parameter for indexing.
- `sizeI` is the size of the distributed dimension.
- `blkSize` is the block size (needed for `block_cyclic` distribution only).

`dist_2D(A, distTypeI, i, sizeI, blkISize, distTypeJ, j, sizeJ, blkJSize)`

- `dist_2D` distributes the first two dimensions of array `A` on 2-D grid of processors.
- `blkISize` is needed when `distTypeI` is `block_cyclic`; and `blkJSize` is needed when `distTypeJ` is `block_cyclic`.

Figure 2.3: Data Distributions in NUMACROS

The above methods only alleviate the performance problems in certain cases. In the next section, we will present NUMACROS, a set of macros, that supports data distributions and parallel loop constructs to avoid mismatching and false sharing remote memory accesses.

2.2 NUMACROS

NUMACROS is a set of C macros for scalable shared-memory machines with Non-Uniform Memory Access time (NUma MACROS) for developing parallel applications using the data parallel model. NUMACROS includes a set of parallel loop constructs that schedule iterations to processors, and a set of data distribution constructs that partition arrays among processors. Our goal in designing NUMACROS is to show that

- Data locality can be obtained by the coordination of loop scheduling and data distribution, so high performance is achievable on scalable shared-memory machines.
- Scalable shared-memory machines can provide better support for implementing data parallelism than message-passing machines.

A parallel program in NUMACROS starts with several threads that execute the `Main` function. To minimize scheduling overhead, the number of threads is equal to the number of allocated processors. These processors are viewed as a one-dimensional grid (1-D) or two-dimensional grid (2-D). The number of processors is chosen at run time as parameter P for a 1-D grid or parameters (P_1, P_2) to form a $P_1 \times P_2$ 2-D grid.

Variables in NUMACROS are either private or shared. Local variables defined within functions are private, so each thread has its own copy. Global variables defined outside of functions are shared by all threads. Scalar shared variables reside in the processor that starts to run the program, but arrays can be distributed among processors to improve locality and to reduce communication contention.

2.2.1 Data Distributions

NUMACROS currently supports three types of of regular data distributions, `block, cyclic`, and `block_cyclic`, for partitioning an array dimension among processors. Two macros, `dist_1D` and `dist_2D`, are defined for mapping arrays onto a 1-D grid and a 2-D grid of processors, respectively. The parameters of `dist_1D` and `dist_2D` are given in Figure 2.3.

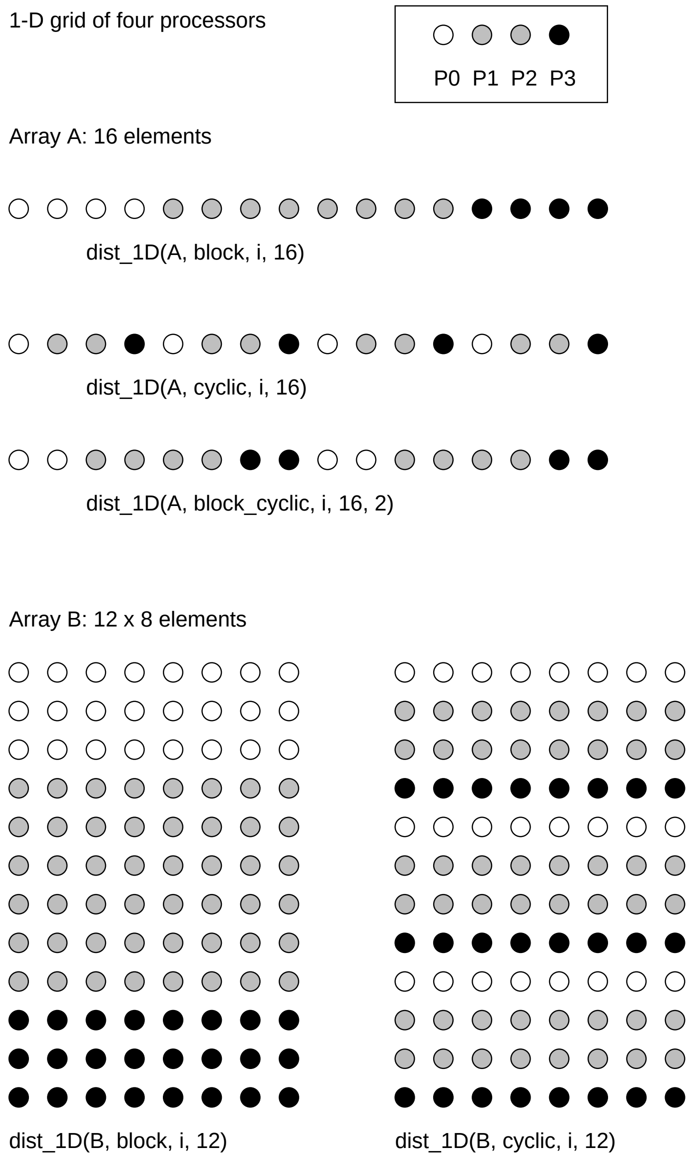


Figure 2.4: Examples Using dist_1D

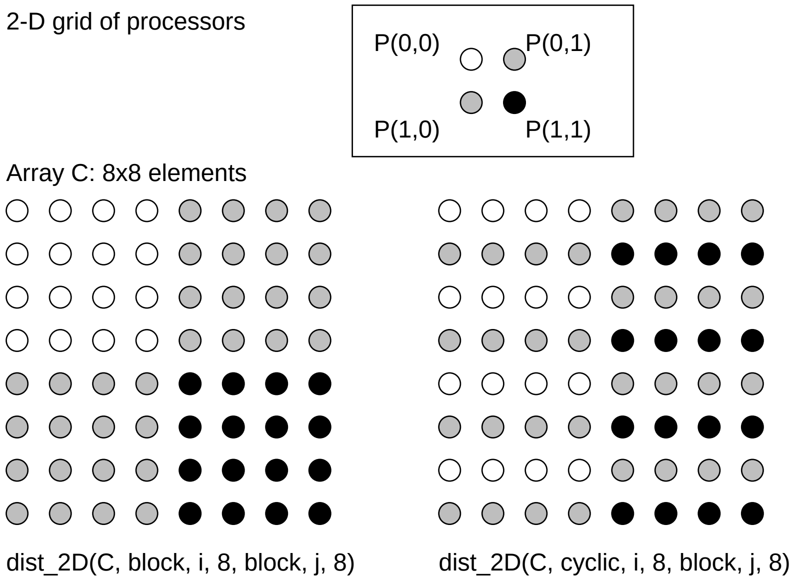


Figure 2.5: Examples Using dist_2D

Figure 2.4 shows how a one-dimensional array and a two-dimensional array can be distributed on a 1-D grid of processors in a *block, cyclic*, or *block-cyclic* fashion [FHK+90, KM91]. The choice of distribution type depends on communication patterns. Block distribution reduces communications if many references are made to neighboring elements. Cyclic distribution is often useful for load balancing if only a subrange of the original array will be used. Block-cyclic distribution is a compromise between block and cyclic patterns, and is often used to trade off between load balancing and data locality.

The macro `dist_2D` provides block, cyclic, and block-cyclic data distributions for a 2-D grid of processors. For example, `dist_2D(A, block, i, N, cyclic, j, N)` specifies that the rows of array `A` are distributed in blocks along the first dimension of the processor grid and the columns are distributed in cyclic fashion along the second dimension of the processor grid. Examples of the use of `dist_2D` are given in Figure 2.5.

2.2.2 Parallel Loops

In NUMACROS, a number of parallel loop constructs are defined to match the data distributions, so that good locality can be achieved. For a 1-D grid of processors, NUMACROS provides parallel loops of `do_blk,` `do_cyc`, and `do_blk_cyc`, which schedule iterations of loops in cyclic, block, and block-cyclic fashions, respectively [KM91]. Parameters of these macros are given in Figure 2.6.

To reduce the communication cost in applications based on multi-dimensional grids of data, NUMACROS allows data and loop nests to be mapped onto a 2-D grid of processors. The approach can be extended to accommodate higher dimensions. On a 2-D grid of processors, parallel loop constructs `do1_blk, do1_cyc`, and `do1_blk_cyc` partition the iterations of loops along the first dimension of the grid, and parallel loop constructs `do2_blk, do2_cyc`, and `do2_blk_cyc` partition them along the second dimension of the grid. For example, `do1_cyc(i, l, u)` indicates that each processor on row p_1 of the 2-D processor grid ($P_1 \times P_2$) will execute iterations from $p_1 * (u-l)/P_1$ to $(p_1+1) * (u-l)/P_1 - 1$.

2.2.3 LU Decomposition in NUMACROS

Figure 2.7 (b) shows the parallel version of the LU decomposition algorithm written in NUMACROS. The code is similar to the sequential version in Figure 2.1 (a) except for a construct for data distribution, `dist_1D`, and a construct for a parallel loop, `do_cyc`. This algorithm has a sequential outer loop, a parallel loop, and a sequential inner loop. Since the iteration space of parallel loop j reduces with each successive iteration of the outermost loop k, the `do_cyc` loop is used to schedule iterations among processors in cyclic fashion for

- `do_cyc(i, l, u)` schedules the iterations ($l \leq i < u$) of the loop in cyclic fashion, which means iteration `i` will be executed by thread `(i mod P)`. Parameter `i` is a private variable, and `l` and `u` are the upper and the lower bounds of loops.
- `do_blk(i, l, u)` partitions the iterations ($l \leq i < u$) of the loop into `P` blocks, and each thread executes one block.
- `do_blk_cyc(i, l, u, blksize)` partitions the loop into blocks of size "`blksize`", and schedules the blocks in a cyclic fashion.

Figure 2.6: Parallel Loops in NUMACROS

```
double A[N][N];

main()
{  int i, j, k;
   ... ...
   for (k = 0; k<N-1; k++)
     do_cyc(j=k+1; j<N; j++)
       A[j][k] /= A[k][k];
       for ( i = k+1; i<N; i++)
           A[j][i] -= A[j][k]*A[k][i];
     enddo
}
     (a) UMA version
```

```
double A[N][N];
dist_1D(A, cyclic, i, N)
Main()
{  int i, j, k;
   ... ...
   for (k = 0; k<N-1; k++)
     do_cyc(j, k+1, N)
       A(j,k) /= A(k,k);
       for (i = k+1; i<N; i++)
         A(j,i) -= A(j,k)*A(k,i);
     enddo
}
(b) NUMACROS version
```

Figure 2.7: LU Decomposition in NUMACROS

load balancing. In C, the row-major storage method for arrays is used. If the row size of an array is not a multiple of the physical memory page size, page false sharing occurs between threads operating on adjacent rows. To reduce such false sharing, NUMACROS provides data distribution constructs that map arrays onto processors properly, observing physical memory boundaries. In this example, `dist_1D(A, 1, cyclic, i, N)` is used to distribute rows of array `A` in a cyclic fashion among processors, so that accesses to all rows, except the pivot row, are local. Note that references to array `A` between `do_cyc` and `enddo` in Figure 2.7 (b) resemble array references with subscripts, but actually are macros with parameters.

As shown in this example, the NUMACROS version of LU decomposition is nearly as simple as the sequential version. With the support of NUMACROS, programmers do not need to know the details of the memory management and processor scheduling policies in operating systems and the cache coherence problem. NUMACROS provides an efficient data layout, and programmers still use the same notations for array references as if these arrays were stored in the fashion the C language defines. Therefore, they can concentrate on algorithm design and program development.

2.3 Implementation of NUMACROS

In this section, we describe the implementation of loop partitioning and data distribution in NUMACROS on Hector [VSWL91]. Appendix A lists C macros that implement data distributions and parallel loops on a 1-D grid of processors for Hector.

```
#define cyc_macros(i, a, b, pid, P) { int _start = ((a)/(P))*(P) + pid;  \
                          if (_start <(a)) _start += (P); \
                          for (i= _start; i<(b); i += (P)){

#define blk_macros(i, a, b, pid, P) { register _start, _end; \
                _end = (b- (a))/(P) + ((b- (a))% (P)>0); \
                _start = a + _end*(pid) ; \
                _end = min(a+ _end*(pid+1), b);\
                for (i = _start; i<_end; i++) {

#define blk_cyc_macros(i, a, b, B, pid, P) { for (i=a, i<b; i++) \
                if (pid == (i/(B) % (P))) {

#define do_cyc(i, a, b) cyc_macros(i, a, b, myPid, P)

#define do1_cyc(i, a, b) cyc_macros(i, a, b, myP1, P1)

#define do2_cyc(i, a, b) cyc_macros(i, a, b, myP2, P2)

#define do_blk(i, a, b) blk_macros(i, a, b, myPid, P)

#define do1_blk(i, a, b) blk_macros(i, a, b, myP1, P1)

#define do2_blk(i, a, b) blk_macros(i, a, b, myP2, P2)

#define do_blk_cyc(i, a, b, blkSize) blk_cyc_macros(i, a, b, blkSize, myPid, P)

#define do1_blk_cyc(i, a, b, blkSize) blk_cyc_macros(i, a, b, blkSize, myP1, P1)

#define do2_blk_cyc(i, a, b, blkSize) blk_cyc_macros(i, a, b, blkSize, myP2, P2)

#define enddo   }; barrier;}

#define end }}
```

Figure 2.8: Parallel Loops in NUMACROS

2.3.1 Parallel Loops

A parallel loop can be scheduled on all processors or on one dimension of a 2-D grid of processors. The run time parameters for processor configurations are P: the total number of processors, or P_1 and P_2: dimension sizes of a 2-D grid of processors (where $P = P_1 \times P_2$). Each thread/processor is assigned an identifier (`myPid`) on the 1-D grid, or an identifier pair (`myP1, myP2`) on the 2-D grid, which are used for scheduling parallel loops.

The implementation of parallel loops in C macros are given in Figure 2.8. As an example, we describe here cyclic parallel loops. In `do_cyc(i, a, b)`, the iteration set on processor `myPid` is defined as $\{i \mid (i \bmod P = myPid) \;\&\; (a \leq i < b)\}$, while in `do_cyc1(i, a, b)`, all processors with an identifier pair *(myP1, X)* (where $(0 \leq X < P_2)$) will execute the same iteration set: $\{i \mid (i \bmod P_1 = myP1) \;\&\; (a \leq i < b)\}$. At the end of a parallel loop, all threads are synchronized with a barrier. If the execution times of iterations of a parallel loop have high variance, fine-grain synchronization primitives (such as locks, or process counters) may be more efficient than barriers. For regular parallel loops, we observed that the performance difference between using barriers and using locks was negligible on Hector.

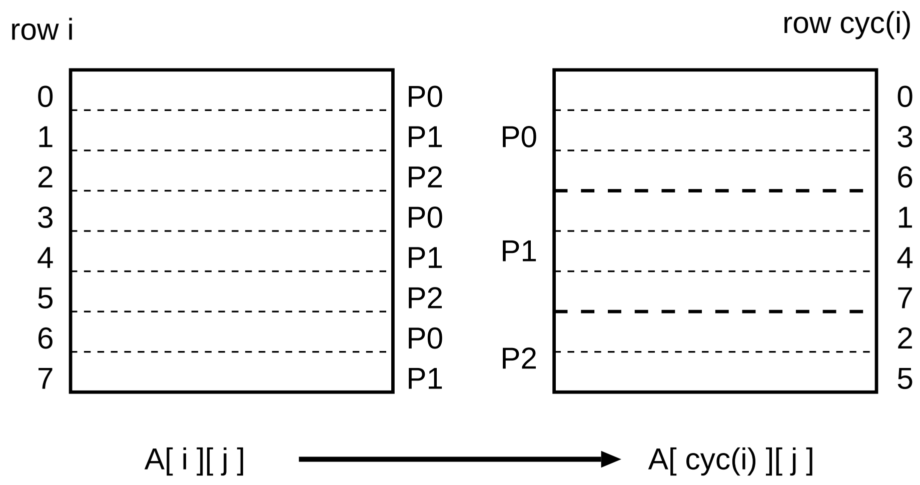


Figure 2.9: Cyclic Distribution Mapping on 1-D Grid Processors

2.3.2 Data Distributions

Data distribution schemes on message-passing multiprocessors partition arrays into smaller subarrays, each of which is assigned to one processor. These subarrays are stored in different address spaces. On scalable shared-memory machines, since the shared-memory paradigm is supported by hardware, we can define good data layouts in the global address space for data distributions. To implement data distribution on scalable shared-memory machines, our approach is to map original arrays onto other arrays with good data layouts. We discuss row-cyclic/row-block distributions on a 1-D grid of processors and block distributions on a 2-D grid of processors. Our approaches can be applied to other regular data distributions as well.

Block distribution on a 1-D grid. Consecutive rows are mapped onto pages and these pages are placed in processor memories based on a block distribution. All pages that contain only array elements accessed by processor i under the block distribution will be mapped to processor i's memory. Also, processor i may share one page with processor $i+1$. This false sharing is negligible if each processor has many rows. Hence, this distribution method can rely on the First-Hit page placement policy in the operating system.

Cyclic distribution on a 1-D grid. False-sharing may exist in every row if the row size is not equal to a multiple of the page size. To reduce false sharing, cyclic rows (i, $i+P$, $i+2P$, $\ldots$) in the original array are mapped onto consecutive ones so that processors i and $i+1$ may share at most one page. Such a mapping, denoted $cyc(i)$, is illustrated in Figure 2.9 for the case of mapping 8 rows onto 3 processors cyclically.

Block distribution on a 2-D grid. If an application requires 2-D block-partitioning of an array to reduce the communication volume between processors, then row-major (or column-major) storage of the array leads to substantial page false sharing and thus increases the number of remote memory accesses.(See Figure 2.10 (a).) One approach to eliminating false sharing is to tile the two-dimensional array into a four-dimensional array based on the grid of $P_1 \times P_2$ processors. For example, array `A[N][N]` is mapped to `A[`P_1`][`P_2`][`S_1`][`S_2`]`, where $P_1 \times S_1 = N$, $P_2 \times S_2 = N$, and each sub-array of size $S_1 \times S_2$ will be allocated to one processor's local memory. Since row-major storage is used by C compilers, only the second dimension needs to be tiled. So the original array `A[N][N]` is stored as `A[`P_2`][N][`S_2`]`. The reference to `A[i][j]` in the original array in Figure 2.10 (a) is replaced by the reference to `A[blk(j)][i][off(j)]` in the transformed array in Figure 2.10 (b), where *blk(j)* is the block number on the j dimension and *off(j)* is the offset within a block.

2.3.3 Shared-Memory versus Message-Passing

Shared-memory multiprocessors make the implementation of data parallel programming much easier than message-passing multiprocessors. On message-passing machines, data distribution is necessary to obtain parallelism, so message-passing operations must be generated *precisely* according to the data distribution.

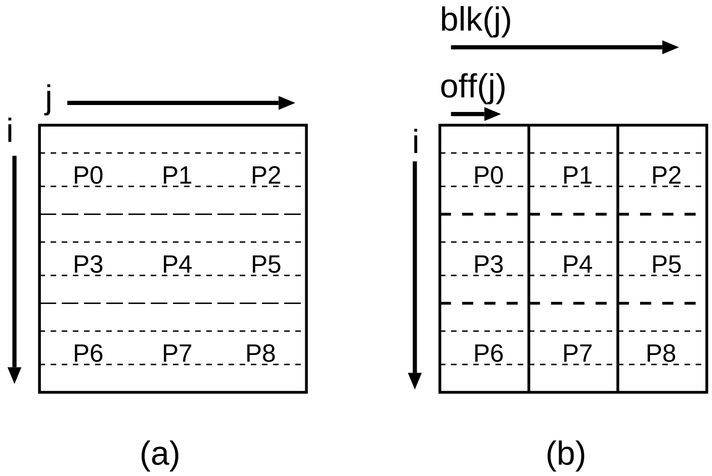


Figure 2.10: Block Distribution Mapping on 2-D Grid Processors

On shared-memory machines, data distribution is used to retain data locality, so transformed arrays are defined to *approximately* match the desired data distribution among processors.

- The owner-computes rule. On message-passing machines, this rule simplifies the generation of message-passing primitives. However, it adds extra constraints on loop scheduling and data alignment. In the following example,

```
for (i=a; i<b; i++) {
    A[i]   = ...
    B[i-1] = ...
}
```

  if `A[i]` and `B[i-1]` are not distributed on the same processor, then the owner-computes rule requires the two statements within the same iteration to be executed on the processors which own `A[i]` or `B[i-1]` respectively. On shared-memory machines, we can just partition iterations based on the block data distribution among processors, without the need for considering on exactly which processor each array element is located. The loss of data locality in such an approximate approach is negligible.

- Data alignment. For the above example, data alignment can also be used to place `A[i]` and `B[i-1]` on the same processor. This is important in message-passing multiprocessors, but may not always be useful for shared-memory machines because the performance benefit from redistributing one of the arrays may not outweigh the overhead.

- Array references. To access non-local array elements on message-passing machines, we need to generate message-passing primitives to get the elements from other processors and we must allocate a temporary array to hold the elements. Then, for each array reference, we need to check whether it will access the local array or the temporary array. If this cannot be determined at compile time, guard conditions must be added for those references, resulting in some extra overhead. On shared-memory machines, we can directly access transformed arrays.

- Irregular applications. Accesses to sparse arrays require additional overhead for message-passing machines. Consider sparse matrix-vector multiplication, $\mathbf{x} = A\mathbf{b}$, where we partition the vector $\mathbf{x}$ and the rows of the sparse array A using a block distribution. On shared-memory machines, we partition $\mathbf{b}$ in blocks as well, and the accesses to $\mathbf{b}$ depend on the structure of the sparse array A. On message-passing machines, compilers must insert some code that, at run-time, generates message-passing operations for accesses to $\mathbf{b}$.

Message-passing machines also require several additional optimizations, such as storage management of non-local elements, message vectorization for reducing communication setup cost, and overlapping communication with computation. These optimizations are not applicable to shared-memory machines. Because data transformation is used to implement data distribution on shared-memory machines, the index computation for transformed arrays becomes more complicated. We will discuss several optimization techniques for index computation in the next chapter.

2.4 Impact of Data Locality

We experimented with four basic programs: Matrix Multiplication (MM), LU Decomposition (LU), Successive Over Relaxation (SOR), and Mean Value Analysis (MVA) on the 16-processor Hector system.

- Matrix Multiplication (MM). The regular matrix multiplication algorithm was parallelized based on the outer loop (shown in Figure 2.11 (a)). The matrices to be multiplied each contained 300×300 double precision numbers.
- LU Decomposition (LU). A matrix of 400×400 double precision numbers was processed. The middle loop was parallelized and scheduled in a cyclic fashion. (See Figure 2.7 (b).)
- Successive Over Relaxation (SOR). SOR is an iterative linear system solver. The unknowns are stored in an array and marked as red and black alternatively in both row and column directions. The first parallel loop updates each red grid element using its previous value and black neighboring elements, and the second parallel loop updates black elements in a similar manner (see Figure 2.11 (b)). In our implementation, the number of unknowns was chosen to be 400×400 and the algorithm stopped after 100 iterations. Because every processor has to access all its neighboring elements, locality plays a major role in obtaining good performance.
- Mean Value Analysis (MVA) [LZGS84]. This algorithm solves a closed queueing network for mean performance values such as average response times, throughputs, and average queue lengths at the queueing centers. A queueing network consists of a number of queueing centers and a number of customer classes. A customer class is defined for each group of customers with identical visit and service patterns. A closed queueing network is one in which the number and composition of customers are fixed. For simplicity, we assume a 2-class closed queueing network with K queueing centers and C customers in each of the two classes. Since the task structure in MVA is diamond shaped, `do_cyc` on rows is used to parallelize the loop to achieve load balancing.

Each application may have the following versions:

- NUMACROS. Data distributions are annotated in NUMACROS based on access patterns. For both MM and SOR, the row-block distribution is used to match the row-based block loop partitioning, and thus good locality is achieved. For LU decomposition and MVA, cyclic loop scheduling is necessary to achieve good load balancing, and thus row-cyclic distribution is used.
- Round-Robin (RR). The operating system's Round-Robin policy is used to place pages in memories of all the processors in the system. Although it might avoid contention, it leads to poor data locality with many remote accesses even for one processor.
- First-Hit (FH). The operating system First-Hit policy is used to place pages in the memories of the processors that use them most. If a page is shared by two or more processors, it will be allocated on one of the processors. Therefore, false sharing may occur.
- Round-Robin No False-sharing (RR-NF). This version also uses the RR policy, but the data size (for example, the array row size) is increased to be a multiple of the page size to reduce false sharing. So some portion of the space will never be used.
- First-Hit No False-sharing (FH-NF). This version also uses the FH policy to obtain relatively good data locality. As in RR-NF, it also increases the data size to reduce false sharing.

```
double A[N][N], B[N][N], C[N][N];
dist_1D(A, block, i, N)
dist_1D(B, block, i, N)
dist_1D(C, block, i, N)
Main()
{  int i, j, k;
   ... ...
   do_blk(i, 0, N)
     for (j=0; j<N; j++)
       for (k=0; k<N; k++)
          C(i,j) += A(i,k)*B(k,j);
   enddo
}
```

(a) Matrix Multiplication

```
double A[N][N];
dist_1D(A, block, i, N)


Main()
{ int i, j, t;
  ... ...
  for (t =0; t<100; t++) {
    do_blk(i, 1, N-1)
        update red grids;
    enddo
    do_blk(i, 1, N-1)
        update black grids;
    enddo
}
```

(b) SOR

Figure 2.11: Matrix Multiplication and Successive Over Relaxation in NUMACROS

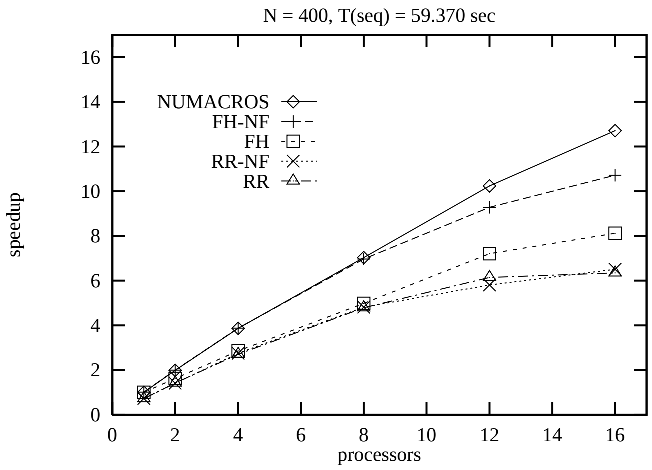


Figure 2.12: Impact of Data Locality in LU Decomposition

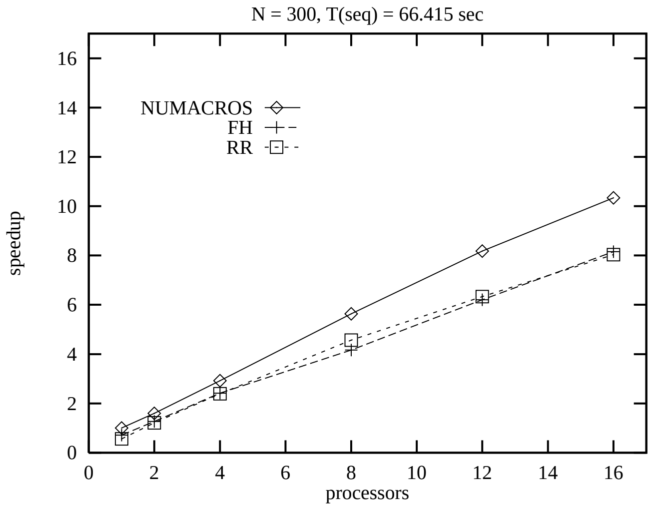


Figure 2.13: Impact of Data Locality in Matrix Multiplication

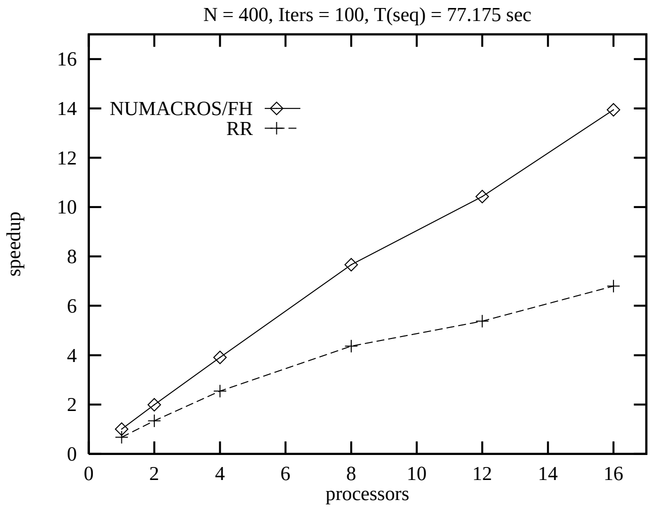


Figure 2.14: Impact of Data Locality in Successive Over Relaxation

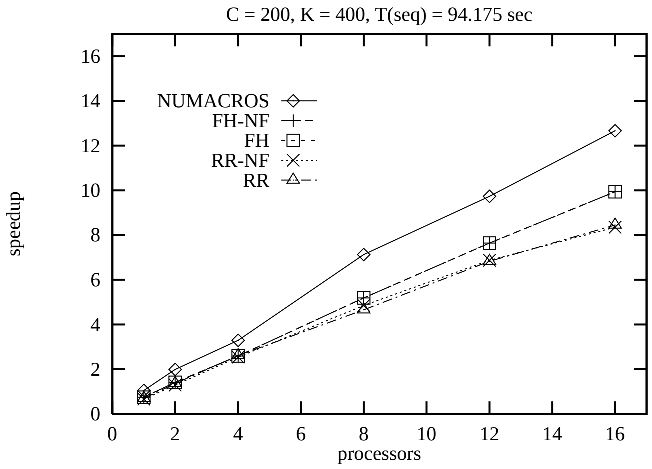


Figure 2.15: Impact of Data Locality in Mean Value Analysis

The speedups of the four applications are shown in Figures 2.12 – 2.15.

LU. The versions with FH outperform the versions with RR because of data locality. With NF, FH-NF further improves performance because NF avoids false sharing on pages. However, RR-NF does not benefit from NF since most references in both RR and RR-NF are remote. NUMACROS not only obtains good data locality in main memory, but also increases the cache hit ratio because data accessed by the processors are mapped to consecutive locations in order to reduce conflict misses.

MM. Since false sharing in MM only appears on boundary rows, the number of false shared pages is negligible (at most P). Thus NF versions are not included. The speedups of RR and FH are very close because they have similar memory access patterns as follows: (1) most accesses to array C are through registers; (2) most accesses to array B are remote; and (3) although accesses to array A are local in FH and remote in RR respectively, they are cacheable. NUMACROS performs better by transposing array B to obtain spatial locality in cache lines.

SOR. False sharing is again negligible. The NUMACROS version is identical to the FH version in SOR. Since RR places pages in a round-robin fashion system-wide, the poor data locality results in lower speedups.

MVA. MVA is both computation and communication intensive. Since the iterations are scheduled in a round-robin fashion, the data produced by processor p will be used by processor $(p+1)$. Even though FH eliminates some remote accesses, the number of remote memory accesses is still high. Therefore, the difference between speedups in FH and RR is small, and the effect of NF is not noticeable. The improvement in NUMACROS is due to the higher cache hit ratio.

From these experments, we have found that

- For applications (such as MM) which inherently require many remote memory accesses, NUMACROS might only slightly improve performance since the number of mismatching and false sharing remote memory accesses only accounts for a small portion of the total number of remote accesses.

- For applications (such as MM and SOR) with negligible false sharing, a careful use of the First-Hit policy can obtain good data locality in main memory. NUMACROS performs at least as well as First-Hit, but might not give much improvement.

- The Round-Robin policy ignores data locality, and therefore does not perform well in most applications, especially in applications that are sensitive to locality (for example, the neighbor access patterns in SOR and MVA).

- For applications (such as LU and MVA) in which both data locality and false sharing are important, NUMACROS significantly outperforms other approaches. Although FT-NF can achieve good data locality in main memory and avoid false sharing, NUMACROS can also obtain a higher cache hit ratio, and thus outperform FT-NF.

While data distributions are essential to scalable NUMA multiprocessors, they are not necessary for the KSR1 because its COMA architecture automatically migrates and replicates data in the cache (local) memory. However, other data transformations may improve data locality on COMA architectures as well (as will be described in Chapter 3).

2.5 Summary

We have proposed and implemented a set of macros (NUMACROS) for data parallel programming on scalable shared-memory machines. Programs written using NUMACROS are nearly as short and easily readable as sequential versions of the programs. Yet, these programs can achieve good speedups since the NUMACROS lead to good data distribution and loop partitioning. Therefore, NUMACROS supports both ease of programming and high performance. We have found that

- With scalable shared-memory systems, data locality is the key to good performance of parallel applications. To obtain data locality, data and loops should be distributed and partitioned among processors in a coordinated fashion.

- Shared memory simplifies the implementation of the data parallel programming paradigm.

Chapter 3

Data Transformation

Multi-dimensional arrays are usually mapped onto the one-dimensional address spaces of physical memories by compilers in either column-major or row-major fashion. Such mappings suffice for sequential machines and small-scale multiprocessors, because the time to access any array element is independent of its location in the memory. However, they are not efficient for scalable shared-memory machines, since the physical memory is distributed.

On message passing machines, arrays must be partitioned into sub-arrays in different address spaces across processors. Hudak and Abramham [HA90] develop algorithms for partitioning arrays for single iterative parallel loops that often appear in PDE solvers. Ramanujam and Sadyappan [RS91] find a way to obtain communication-free partitioning of arrays in single parallel loops. For multiple parallel loops, Li and Chen [LC90, LC91] consider aligning array dimensions for reducing the amount of communication, and Anderson and Lam [AL93] consider both data alignment and parallelism. On scalable shared-memory multiprocessors, since all memory locations are accessible by all processors, arrays need not be partitioned into sub-arrays. Data mapping on such multiprocessors thus involves many different issues such as false sharing and data reuse in local memory and cache.

While loop transformations have been extensively studied for improving parallelism and locality, only a few data transformations (such as array transpose and array fusion) have been commonly used as hand-tuning techniques for improving cache locality on uniprocessors. In this chapter, we propose a general framework of data transformations and describe their uses in data locality optimization. This framework is complementary to loop transformation theory, and provides a basis for further studies for unifying data transformations and loop transformations [1].

In Chapter 2, we have proposed several data transformations for effectively implementing regular data distributions for scalable shared-memory multiprocessors. In this chapter, we further propose a set of data transformations, classify them into unimodular and non-unimodular transformations, and describe their uses in data locality optimization. Since the transformed arrays require more complicated index expressions, we derive several new techniques for optimizing the computation of such index expressions. To improve spatial cache locality, we propose a new global optimization algorithm that produces unimodular data transformations to reorder array dimensions. Finally, we show how these data transformations can be applied for improving cache locality and reducing the number of remote memory accesses.

3.1 Types of Data Transformations

A data transformation is a mapping from an original array A to another array A'. Every reference to array A will be translated to a reference to array A'. The goal of the transformation is that accesses to the transformed array A' cost less than accesses to A on average. A simple example is the transpose of a two-dimensional array. If an array that is stored row-wise is primarily accessed column-wise, then storing

[1] Michal Cierniak and Wei Li have recently been working on unifying linear data and control transformations [CL94]. Their work focuses on single array references.

Data Transformations	Loop Transformations	Optimization
unimodular (e.g., transpose)	unimodular transformations (e.g., interchange)	improve locality implement data distribution
block, cyclic	tiling	implement data distribution improve locality reduce false sharing
partitioning	distribution	reduce communication
fusion	fusion	block transfer, prefetch
scaling	scaling	reduce communication

Table 3.1: Overview of Data Transformations

the transposed array can provide a better cache hit ratio.

We propose a set of data transformations (see Table 3.1). There are strong similarities between loop transformations and data transformations. For example, loop interchange reorders loops in the loop nest; similarly, array transpose reorders array dimensions. The legality of loop transformations is constrained by the data dependences, but the utility of data transformations depends only on the performance benefit. In this section, we describe both unimodular and non-unimodular transformations, and discuss their uses in locality optimization.

3.1.1 Unimodular Data Transformations

A unimodular transformation on a d-dimensional array A is defined by :
U : a $d \times d$ unimodular transformation matrix
$\mathbf{N}$: a modular vector
References to A by the index vector $\mathbf{I}$ are replaced by references to the transformed array A' through the index vector: $U\mathbf{I} \bmod \mathbf{N}$.

The unimodular transformation matrix controls the orientation of transformed arrays, and the modular vector wraps a parallelogram index set into a rectangle. A unimodular matrix is an integer matrix with its determinant value equal to one.

Permutation

The permutation of an array dimension can be defined by a unimodular matrix. For example,

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

maps references to array $A[i][j][k]$ to $A'[j][k][i]$. Like loop reordering [Car93], array permutation can significantly improve cache locality. For example, in the loop

```
for (i = 0; i<N; i++)
   for (j = 0; j<N; j++)
      sum += A[j][i];
```

elements `A[j][i]` and `A[j+1][i]` are not stored close to one another since array `A` is stored in row-major order. Therefore, the spatial locality within cache line cannot be exploited. The temporal locality of reusing cache lines between the iterations of loop `i` cannot be captured, either because of the large amount of data or because of cache line conflicts. Both loop interchange and array transpose can solve this problem as shown in these code fragments:

```
for (j = 0; j<N; j++)          for (i = 0; i<N; i++)
   for (i = 0; i<N; i++)          for (j = 0; j<N; j++)
      sum += A[j][i];                sum += A'[i][j];

(a) loop interchange            (b) array transpose
```

However, for some loop nests, loop interchange may not be legal due to data dependences. Besides, if a loop nest contains references to several different arrays, loop interchange may not make all reference sequences be in row-wise order in the innermost loop.

The permutation of array dimensions is always applicable, but has an associated cost for transforming an array into an appropriate form. We will present algorithms for selecting array permutations in Section 3.3.

Skewing

Unimodular loop transformation unifies loop interchange, reverse, and skewing. Similarly, unimodular data transformation integrates array permutation and skewing. Skewed arrays can avoid data redistribution (as Shindo *et al.* also observed [SIK+94]).

In the following loop nests,

```
doall (i = 0; i<N; i++) {
   ... ...
   for (j = 0; j<N; j++)
      A[i][j] = f(A[i][j-1]);
}

for (i = 0; i<N; i++) {
   ... ...
   doall (j = 0; j<N; j++)
      A[i][j] = f(A[i-1][j]);
}
```

neither row-block distribution nor column-block distribution among processors can satisfy both of the `doall` loops. This dilemma can be solved by a skewing transformation defined as

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mathbf{I} \bmod \begin{bmatrix} N \\ \infty \end{bmatrix}$$

The skewed array is shown in Figure 3.1, where the unimodular matrix shapes the square array in Figure 3.1 (a) into a parallelogram (Figure 3.1 (b)), and the unimodular vector wraps the parallelogram into a square array (Figure 3.1 (c)) to save memory space.

3.1.2 Non-unimodular Data Transformations

Orthogonal to a set of non-unimodular loop transformations, we propose a set of data transformations that cannot be represented by unimodular matrices. They include regular data distribution, partitioning, fusion, and scaling.

Cyclic and Block Distributions

The data transformations for cyclic and block distributions have been illustrated in Figures 2.9 and 2.10. For a row-cyclic distribution of a two-dimensional array `A` of size $N \times N$ among P processors, the row-cyclic

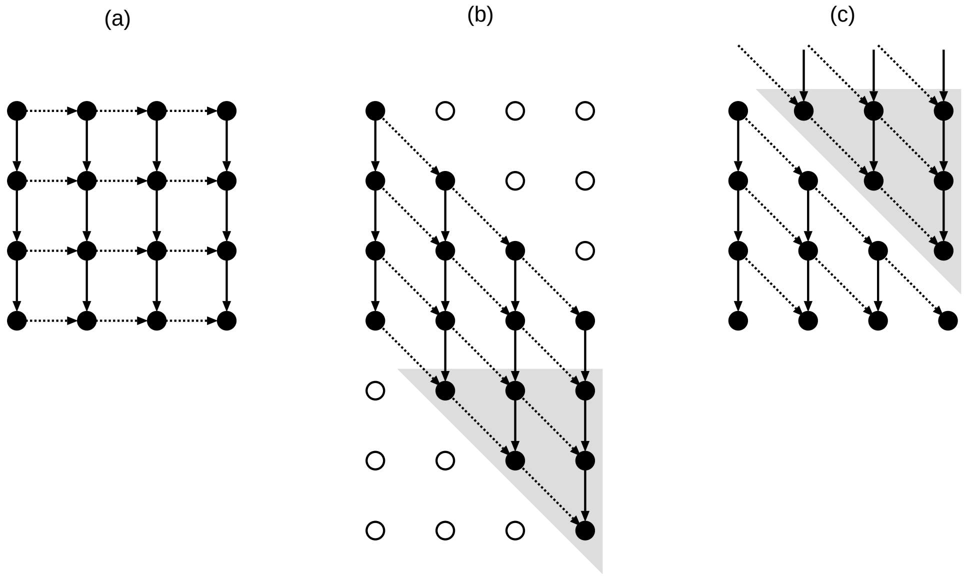


Figure 3.1: Skewed Array

transformation maps reference `A[i][j]` to `A'[cyc(i)][j]`, where the transformed matrix size is $\lceil N/P \rceil \times P$ (which can be slightly larger than N). The mapping function is

$$cyc(i) = \lceil N/P \rceil \times (i \bmod P) + \lceil i/P \rceil$$

For the block distribution on the 2-D ($P_1 \times P_2$) grid, a transformed array is tiled on the second dimension with the block size of S_2, so mapping functions from indexing the original array to indexing the transformed array by block numbers and offsets within the blocks are

$$blk(j) = \lceil j/S_2 \rceil, \qquad off(j) = j \bmod S_2$$

References to the original `A[i][j]` are thus replaced by `A[blk(j)][i][off(j)]`.

The main purpose of data distribution is to reduce mismatching remote accesses and false sharing accesses as we have discussed in Chapter 2. The data transformations we propose for implementing data distribution also reduce cache conflict misses, because the array elements that each processor uses most are mapped onto contiguous addresses.

Fusion

Loop fusion merges several (parallel) loops into one to increase granularity. Data fusion combines several arrays into one to increase data block size so that block transfer or prefetch can be effectively applied. In Figure 3.2 (a), the elements `A[i], B[i], C[i]`, and `D[i]` are not stored adjacently. In Figure 3.2 (b), we apply fusion to merge arrays `A, B, C`, and `D` into a record array `R`. To perform prefetch on these loops, we need to insert four prefetch instructions in Figure 3.2 (a), but only one prefetch instruction in Figure 3.2 (b). In addition, fusion can also reduce the number of cache conflict misses.

Partitioning

Data partitioning is the opposite of data fusion, as loop distribution is the opposite of loop fusion. It splits an array into a few smaller ones in order to reduce the number of remote memory accesses and cache misses in a loop with a stride greater than one.

Figure 3.3 shows how data partitioning is applied to Red-Black SOR. Originally, the rows of array `A` are distributed in blocks among three processors. Each white element updates itself using its neighboring black elements, and vice versa. The white elements on the first row on processor 2 need to accesses black elements

```
double A[N], B[N], C[N], D[N];        struct {
                                        double A, B, C, D;
                                      } R[N];

for i = 0 to N-1                      for i = 0 to N-1
   D[i] = f(A[i], B[i], C[i]);           R[i].D = f(R[i].A, R[i].B, R[i].C);

          (a)                                      (b)
```

Figure 3.2: Data Fusion

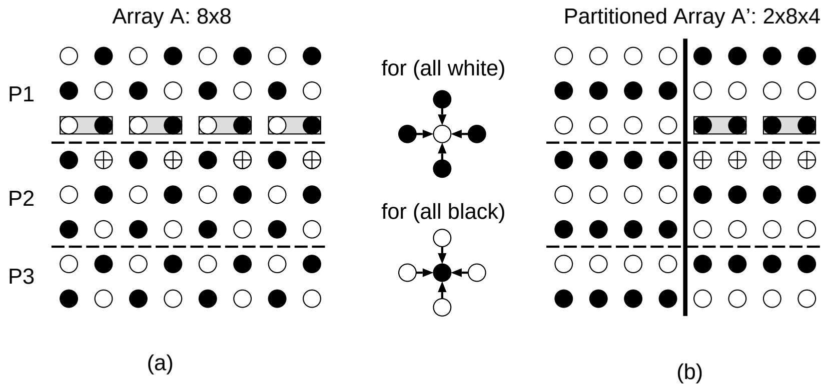


Figure 3.3: Data Partitioning

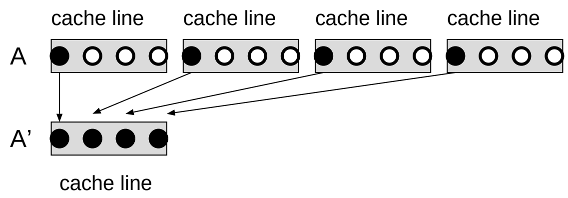


Figure 3.4: Data Scaling

on processor 1. Since the cache line size is usually larger than one word (`e.g.`, two double words on Hector), the adjacent white elements are also brought from processor 1. To avoid such unnecessary accesses to white elements, we partition array `A` is into two parts: odd numbered columns map onto the left side of `A'` and even numbered columns map onto the right side. This is similar to the data transformation for column cyclic distribution among two processors. For computing the white elements of the first row on processor 2, the number of remote memory accesses is reduced from four cache lines in Figure 3.3 (a) to two cache lines in Figure 3.3 (b). The working set before the transformation is three rows: the last row on processor 1 and first two rows on processor 2 (as shown in Figure 3.3 (a)). After the transformation, it is reduced to two rows: the black elements of the last row on process 1, the first row and the black elements of the second row on processor 2 (as shown in Figure 3.3 (b)).

Scaling

Scaling is another transformation that can eliminate unnecessary remote accesses due to the effect of large cache lines. A scaling factor controls the transformed array size. A small scaling factor (< 1) selects some array elements into the transformed one (as shown in Figure 3.4. A larger scaling factor (> 1) scatters array elements from the same cache line into different ones.

All these non-unimodular transformations can be combined with unimodular transformations to obtain better locality. For example, we can combine skewing with regular data distributions to avoid unnecessary data redistribution. We can also apply permutation to find the best order of array dimensions and then use non-unimodular transformations (such as partitioning and data distributions) to reduce both the number of remote memory accesses and the number of cache misses.

3.2 Optimizing Index Computation

After data transformation, references to original arrays in the programs must be replaced by references to the transformed arrays. A naive implementation is to directly use the mapping functions in closed form as shown in the last section. However, the index functions for many transformations (block, cyclic, skewing, etc.) involve division and modulus, which are slow on current processors. Current compilers cannot reduce the strength of these expensive operations. In this section, we propose several optimization techniques for reducing the computation costs of index expressions. These techniques include *index precomputing, strength reduction*, and *loop transformations*. We describe the techniques and their uses in cyclic and block distributions since the mapping functions in these data distributions are more complicated than those in other data transformations.

3.2.1 Index Precomputing

Computing mapping functions (such as *cyc(i), blk(j)*, and *off(j)*) is usually more expensive than making a local memory access. Thus, each processor can compute mapping functions for each array only once and store the results in local arrays. References to transformed arrays are then made by indirect indexing through these local arrays. We call this approach *index precomputing*.

A row-oriented cyclic mapping on the 1-D grid of processors can be computed and stored in a local pointer array where each element points to a row in a cyclic fashion. For the block distribution on the 2-D grid, both block numbers and offsets of the second dimension are computed and stored in two arrays

```
for (i = L; i<U; i++) {
    A[i][...] = ...
    ...
}
    (a) original code

iii = cyc(L); BSize = N/P;
for (p = 0; p<P; p++) {
    for (i = p+L, ii=iii; i<U; i+= P, ii++) {
        A[ii][...] =  ...
        ...
    }
    iii += BSize; if (P == ((L%P)+p+1)) iii++;
    if (iii >= N) iii -= N;
}
    (b) transformed by loop partitioning
```

Figure 3.5: Loop Partitioning

(`blk` and `off`) in the local memory on each processor. A reference to the original `A[i][j]` is replaced by `A[blk[j]][i][off[j]]`.

3.2.2 Strength Reduction

Index precomputing increases the number of local memory accesses. An alternative is to reduce operator strength in the index mapping function by computing increments.

Cyclic distribution on a 1-D grid. Consider a reference with index function, $f(i) = a * i + C$, on the cyclic-distributed dimension on a 1-D grid of processors, where a is a positive integer, i in $f(i)$ is the innermost loop index variable, and C represents the remainder that does not contain loop index variable i, and the stride of loop i is constant s. Then the function $cyc(f(i))$ can be incrementally computed as follows:

$$cyc(f(i+1)) = \begin{cases} cyc(f(i)) + a \times s \times S & \text{if } cyc(f(i)) + a \times s \times S < N \\ cyc(f(i)) + a \times s \times S - N + 1 & \text{otherwise} \end{cases}$$

Where S is the block size ($\lceil N/P \rceil$). This requires only one addition and one comparison per iteration since $a \times s \times S$ needs to be computed only once in the entire loop.

Block distribution on a 2-D grid. The original array `A[N][N]` is stored as `A[`P_2`][N][`S_2`]` where $N = P_2 \times S_2$. For a reference to the original array `A[i][`f_2`(j)]` with $f_2(j) = a * j + C$ where j is assumed to be the innermost loop index with constant stride s and C represents the rest of the invariant in loop j, both block numbers and offsets for the transformed array can be computed incrementally in the innermost loop:

$$blk(f_2(j+1)) = \begin{cases} blk(f_2(j)) & \text{if } \mathit{off}(f_2(j)) + a \times s \le S_2 \\ blk(f_2(j)) + 1 & \text{otherwise} \end{cases}$$

$$\mathit{off}(f_2(j+1)) = \begin{cases} \mathit{off}(f_2(j)) + a \times s & \text{if } \mathit{off}(f_2(j)) + a \times s < S_2 \\ \mathit{off}(f_2(j)) + a \times s - S_2 + 1 & \text{otherwise} \end{cases}$$

3.2.3 Loop Transformations

The strength reduction approach still requires several cycles for index computation. We apply loop partitioning and loop splitting [Wol89b, Wol91] to optimizing index computation and show that they can further reduce this cost.

	Row-Cyclic on 1-D	Cost	Block on 2-D	Cost
Naive	*cyc(f(i))*	`mul`, `div`, `mod`	*blk(f2(j)), off(f2(j))*	`mul`, `div`, `mod`
Precomp.	index array `cyc[N]`	local access	index arrays `blk[N]`,`off[N]`	local accesses
Reduction	incrementally computing *cyc(f(i))*	`add`, `comp`	incrementally computing *blk(f2(j))*, and *off(f2(j))*	`add`, `comp`
Transform.	Part.	add	Part. & Split.	`add`

Table 3.2: Approaches to Computing Indices in Transformed Arrays

Loop partitioning for a cyclic distribution on a 1-D grid. Assume that a loop does not have any loop-carried data dependences [AK87] and that the number of iterations in the loop is greater than the number of processors (P). To optimize the index computation in a reference, the loop can be partitioned into a two-level loop nest in a cyclic fashion such that the reference in the inner loop only accesses data from local memory. If the loop contains multiple references with different index expressions, we choose one of them. Figure 3.5 illustrates the loop partition transformation based on reference `A[i][j]`, but the method can be extended to any linear index expressions (*e.g.*, $f(i) = a*i+C$). Loop i in the original code (Figure 3.5 (a)) is partitioned into an outer loop (on p) and an inner loop i so that only one `add` is needed for indexing (Figure 3.5 (b)). Moreover, since the inner loop accesses consecutive data, the loop partition may yield better spatial locality.

Loop partitioning for a block distribution on a 2-D grid. Similar to loop partitioning on a 1-D grid, a loop is partitioned into a two-level nest so that some references in the inner loop always access a single block of the array. An example of loop partitioning for index optimization is shown in Figure 3.6. Loop j in Figure 3.6 (a) is partitioned into an outer loop (bb) and an inner loop (j) in Figure 3.6 (b). So reference `A[bb][i][jj]` (*i.e.*, the original `A[i][j]`) in the inner loop accesses block bb, requiring one operation (`add`) for indexing. However, references of the form `A[blk(j-1)][i][off(j-1)]` (*i.e.*, original `A[i][j-1]`) have not been optimized since they access more than one block.

Loop splitting for block distribution on a 2-D grid. For index expressions that differ by a constant from the one optimized in loop partitioning (*e.g.*, `A[i][j-1]` and `A[i][j+1]` in the example), loop splitting can be applied. The first and last few iterations are split from the inner loop so that all these index expressions can be computed in one cycle in the inner loop (Figure 3.6 (c)).

Table 3.2 summarizes the approaches to translating references to transformed arrays under the row-cyclic distribution on a 1-D grid of processors and the block distribution on a 2-D grid of processors. Since on current machines, division and modulus are much slower than local memory access, which is in turn slower than addition and comparison, these approaches are ranked in the order of loop transformation, strength reduction, precomputing, and naive with respect to their effectiveness in reducing the cost of indexing.

3.2.4 Effects of Indexing in Transformed Array References

To quantitatively evaluate the proposed optimizations, we implemented three kernel programs, matrix multiplication (MM), LU decomposition (LU), and Jacobi on Hector. Similar to SOR, the Jacobi method makes the new value of each element be the average of the old values of its neighboring elements, and iterates this computation for a number of iterations. In Jacobi, the problem we chose has 300×300 unknowns stored in a 2-dimensional array. It stops after 100 iterations. Both MM and LU use row-cyclic distributions on a 1-D grid of processors, and Jacobi uses a block distribution on a 2-D grid of processors. We applied four indexing approaches (naive, precomputing, strength reduction, and loop transformation) to these programs, resulting in four versions.

The execution times (in seconds) of the three programs on Hector using up to 16 processors are shown in Table 3.3. The execution time ratios (relative to loop partitioning and/or loop splitting transformations) are also given in parentheses. The naive version of each of three programs performed very poorly. The ratios of execution times of the naive version to the loop transformed version are about six in MM and Jacobi, and about three in LU. The index precomputing approach requires that index computation for the data distribution be done only once in the entire program, so it significantly improves performance. However,

```
double A[N][N];
... ...
for (j = 1; j< N - 1; j++)
    A[i][j] = (A[i][j-1] + A[i][j] + A[i][j+1])/3;
```

(a) original loop

```
double A[N][N];
dist_2D(A, BLOCK, i, N, BLOCK, j, N);
...  ...
jj = off(1);
bb1 = blk(1); bb2 = blk(N-1);
for (bb = bb1; bb<= bb2; bb++) {
    for (j = max(bb*S2, A); j < min( (bb+1)*S2, B); jj++, j++)
        A[bb][i][jj] = (  A[blk(j-1)][i][off(j-1)] + A[bb][i][jj]
                        + A[blk(j+11)][i][off(j+1)])/3;
    jj = 0;
}
```

(b) after loop partitioning

```
double A[N][N];
dist_2D(A, BLOCK, i, N, BLOCK, j, N);
...  ...
jj = off(1);
bb1 = blk(1); bb2 = blk(N-1);
for (bb = bb1; bb<= bb2; bb++) {
    if (jj == 0)
       A[bb][i][jj] = (A[bb-1][i][S2-1]+ A[bb][i][jj]+ A[bb][i][jj+1])/3;

    for (j = max(bb*S2+1, A, 2); j< min((bb+1)*S2 -1, B, N-2); jj++, j++)
       A[bb][i][jj] = ( A[bb][i][jj-1]+ A[bb][i][jj]
                        +A[bb][i][jj+1])/3;

    if (jj == S2)
       A[bb][i][jj] = (A[bb][i][jj-1] + A[bb][i][jj] + A[bb+1][i][0])/3;
    jj = 0;
}
```

(c) after loop partitioning and splitting

	Cost				
	mod	div	mul	comp	add
naive	$3N$	$3N$			
partitioning	$2N$	$2N$	$Blocks$	$Blocks$	N
partitioning and splitting			$Blocks$	$3Blocks$	N

(d) costs in different indexing schemes

Figure 3.6: Loop Transformations for Block Data Distribution

	processors	1	2	4	8	16
MM	Naive	430.27 (6.48)	220.04 (5.93)	117.62 (6.02)	60.67 (5.78)	35.35 (6.31)
(300)	Precomp.	73.89 (1.11)	45.20 (1.22)	29.50 (1.51)	15.79 (1.50)	9.05 (1.62)
	Reduction	74.59 (1.12)	40.71 (1.10)	21.58 (1.10)	12.75 (1.21)	6.85 (1.22)
	Transform.	66.42	37.12	19.55	10.50	5.60
LU	Naive	176.92 (2.99)	88.92 (2.97)	45.17 (2.94)	23.21 (2.75)	14.34 (3.28)
(400)	Precomp.	69.44 (1.17)	34.99 (1.17)	17.96 (1.17)	9.64 (1.14)	5.93 (1.27)
	Reduction	60.22 (1.02)	30.93 (1.03)	16.38 (1.07)	8.90 (1.05)	4.90 (1.05)
	Transform.	59.22	29.93	15.38	8.45	4.67
Jacobi	Naive	796.21 (5.92)	398.15 (5.90)	199.35 (6.19)	99.70 (5.86)	52.45 (5.55)
(300)	Precomp.	150.23 (1.11)	76.53 (1.13)	40.77 (1.27)	22.05 (1.29)	13.49 (1.42)
	Reduction	137.16 (1.02)	69.11 (1.02)	35.71 (1.11)	19.06 (1.12)	10.85 (1.15)
	Transform.	134.28	67.46	32.22	17.03	9.45

Table 3.3: Execution Times Using Different Indexing Strategies (in seconds)

it requires extra local memory accesses, which for 16 processors results in an overhead of 62% in MM, 42% in Jacobi, and 27% in LU. The strength reduction approach further improves performance in all the programs and leads to execution time ratios between 1.05 to 1.22. The loop transformation version uniformly outperforms other versions.

3.3 Optimizing Array Dimension Orders

Kennedy, McKinley, and Carr [KM92, Car93] have shown that the loop order in a loop nest greatly impacts data reuse in cache, and they have proposed a simple algorithm for determining the optimal loop order. However, loop reordering may not always be possible because of data dependences. As shown in Section 3.1.1, permuting array dimensions is a possible way of improving cache locality. Without loss of generality, we assume that arrays are stored in column-major fashion and their dimension orders are independently considered.

In this section, we develop algorithms to determine the optimal array dimension order. We propose a simple model to estimate the data reuse for each array, and use the model to derive the optimal array dimension order in each parallel loop. Then, we develop a global algorithm to determine in which loops array dimensions should be reordered. Finally, we present the results of an experiment on the KSR1.

3.3.1 A Reuse Model

Previous studies have shown that inner loop locality is likely to be captured by the cache because of short distances between reuse points, but cache interference often prevents outer loop reuse. Given these factors, we assume that inner loop reuse will dominate outer loop reuse and will be captured by cache. This assumption allows us to arrange the order of array dimensions based on the innermost loop.

Intuitively, if the innermost loop variable appears in dimension i most, the dimension i should be rearranged as the first dimension. For a k-dimensional array in an n level loop nest, we use a $k \times (n+2)$ matrix to represent data reuse as follows:

$$R = \left[\begin{array}{c|cccccc} & const & l_1 & l_2 & \dots & l_n & var \\ \hline d_1 & r_{10} & r_{11} & r_{12} & \dots & r_{1n} & r_{1(n+1)} \\ d_2 & r_{20} & r_{21} & r_{22} & \dots & r_{2n} & r_{2(n+1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ d_k & r_{k0} & r_{k1} & r_{k2} & \dots & r_{kn} & r_{k(n+1)} \end{array}\right]$$

where $L = l_n, \dots, l_1$ is the loop nest, and l_1 is the innermost loop, and $D = d_1, d_2, \dots, d_k$ is the vector of array dimensions. The values of r_{i0} and $r_{i(n+1)}$ are the numbers of different references where indices in dimension d_i are constant values and other variables respectively. To estimate data reuse along dimension d_i in loop l_j, we use r_{ij} to represent the weighted summation of number of different references which contain the index variable I_j of loop l_j in dimension d_i, but not in dimensions $d_{i+1}, \ldots, d_k$. The weight is $\frac{1}{cs_j}$, where c is the coefficient of I_j in the index expression, and s_j is the stride of loop l_j. The data reuse in dimension d_i is thus represented by a tuple $(r_{i1}, \ldots, r_{i(n+1)})$. Since the inner loops dominate data locality, the value of r_{i1} is the most significant. Therefore, the optimal dimension order under the data reuse cost model is determined by sorting these tuples.

When the size of dimension d_i is small (*e.g.*, < 10) and the value of r_{i0} is close to the dimension size, all the elements along dimension d_i are likely to be used in each iteration of the innermost loop. In this case, we choose d_i as the first dimension instead.

3.3.2 A Global Optimization Algorithm

Consider a loop nest sequence $(L^{(1)}, L^{(2)}, \ldots, L^{(m)})$ with reuse matrices $(R^{(1)}, R^{(2)}, \ldots, R^{(m)})$. The local optimal dimension orders for individual loop nests, $(D^{(1)}, D^{(2)}, \ldots, D^{(m)})$ have been obtained as discussed in Section 3.3.1. In this section, we propose a global algorithm that computes the global optimal dimension orders under the reuse cost model for the loop nests, $(D'^{(1)}, D'^{(2)}, \ldots, D'^{(m)})$.

If all loop nests agree on the same dimension order, then this dimension order is the global optimal one. Otherwise, we must determine whether an array permutation is needed between two consecutive loop nests where the local optimal dimension orders differ. Because the data locality on the first dimension in the innermost loop is the most important one, we reorder array dimensions only for exploiting this data locality. Our simple heuristic approach to forming the global solution consists of the following steps:

1. For the first loop nest $L^{(1)}$, we simply choose $D'^{(1)} = D^{(1)}$.

2. For loop nest $L^{(i)}$ where $1 < i \leq m$, if the local optimal dimension order is the same as the global optimal order for its preceding loop, *i.e.*, $D^{(i)} = D'^{(i-1)}$, the loop next $L^{(i)}$ can use the same dimension order, $D'^{(i)} = D'^{(i-1)}$.

3. When the loop nest $L^{(i)}$ requires an array order different from that for $L^{(i-1)}$, a permutation may be worthwhile only if the following criteria are met:

   - The innermost loop variable in $L^{(i)}$ appears in the references to the array we consider, *i.e.*, the second column of $R^{(i)}$ contains non-zero elements.
   - Loop nests $L^{(i-1)}$ and $L^{(i)}$ require different first dimensions in their optimal orders, *i.e.*, $d_1'^{(i-1)} = p$, $d_1^{(i)} = q$ and $p \neq q$.
   - The improvement of data reuse should more than justify the array permutation cost. This includes (a) temporal locality in successive loop nests if they require the same array order, (b) temporal locality within loop nest $L^{(i)}$ itself if its loop nest level is greater than the number of array dimensions, or (c) spatial locality within the innermost loop if the difference of reuse factors for the innermost loop is at least two $(r_{q1}^{(i)} - r_{p1}^{(i)} > 2)$.

3.3.3 Experimental Results

To evaluate our optimization techniques, we applied the algorithm of Section 3.3.2 to the set of NAS kernels (dnasa7), a part of the SPEC benchmarks:

1. MXM – This subroutine performs a standard matrix product on two input matrices. The subroutine employs a four-way unrolled, outer product matrix multiply algorithm.

2. CFFT2D – This test performs a complex radix 2 FFT on a two-dimensional input array, returning the result in place. The test kernel actually consists of two subroutines that perform FFTs along the first and second dimension of the array, respectively.

3. CHOLSKY – This subroutine performs a Cholesky decomposition on a set of input matrices, which are actually input to the subroutine as a single three-dimensional array.

	Original	Transformed	Ratio
MXM	21.8	20.5	1.06
CFFT2D	177.2	115.1	1.54
VPENTA	502.3	176.3	2.85
BTRIX	123.6	40.5	3.05
GMTRY	455.0	74.0	6.15

Table 3.4: Execution Times of the NAS Kernels with Optimized Dimension Orders (in seconds)

4. BTRIX – This kernel performs a block tridiagonal matrix solution along one dimension of a four-dimensional array.
5. GMTRY – This subroutine sets up arrays for a vortex method solution and performs Gaussian elimination on the resulting array.
6. EMIT – Also extracted from a vortex code, this subroutine creates new vortices according to certain boundary conditions.
7. VPENTA – This subroutine simultaneously inverts three matrix pentadiagonals.

Five out of the seven programs in the NAS kernels require reordering array dimensions to achieve better data locality, but the other two programs remain unchanged. We measured the execution times of both the original version and transformed version of these five programs on one processor of the KSR1 (shown in Table 3.4).

Optimizing array dimension orders improves performance by factors from 1.06 in MXM to 6.15 GMTRY. The improvement in MXM is small because arrays `A` and `C` already have the optimal array dimension orders in the original version. References to array `B` do not contain the innermost loop index `I`, so transposing array `B` slightly increases data locality. The 2-D FFT consists of column and row FFT transformations. We transposed the array for performing row FFT transformation, significantly improving cache spatial locality. In GMTRY, one major array is also transposed in the transformed version. In other programs (VPENTA and BTRIX), some arrays are in the optimal order in the original version, but several arrays need to have their dimensions reordered.

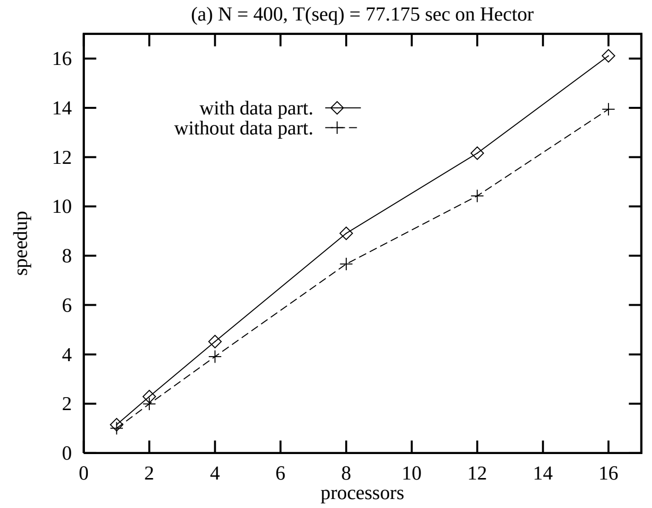
Car *et al.* [Car93, KM92] and Wolf *et al.* [WL91, Wol92] have studied loop transformation for improving cache locality and performed experiments on the NAS kernels. Both studies show that one kernel (MXM) can benefit from loop tiling, and only two others (GMTRY and VPENTA) can benefit from loop interchange. In comparison, optimizing array dimension order gives an improvement in GMTRY comparable to theirs and a higher improvement in VPENTA. While transforming the loops of BTRIX does not improve performance, optimizing array dimension order reduces the execution time by 67%.

On the KSR1, the performance improvement by optimizing array dimension orders is significant, partially due to the large cache line size (64 bytes) and the large primary cache size (256 KB) on the KSR1. On Hector, with a small cache size (16 KB) and a small cache line size (16 bytes), only a marginal improvement can be expected. With increasing cache sizes and cache line sizes on the most recent RISC processors, there will be an even higher performance benefit from optimizing array dimension order.

3.4 Reducing Communications

In Section 3.1, we have shown that several data transformations can reduce communication costs. In this section, we present experimental results in using data partitioning and data fusion transformations.

We applied data partitioning to Red-Black SOR and measured the performance for the 400×400 size problem on Hector and the 512×512 size problem on the KSR1. On Hector, partitioning improves the performance by 15% on 16 processors (Figure 3.7 (a)). On the KSR1, partitioning improves the performance by 10% on 10 processors and by 24% on 30 processors (Figure 3.7 (b)). The higher improvement on the KSR1 is due to the higher remote access time than that on Hector.



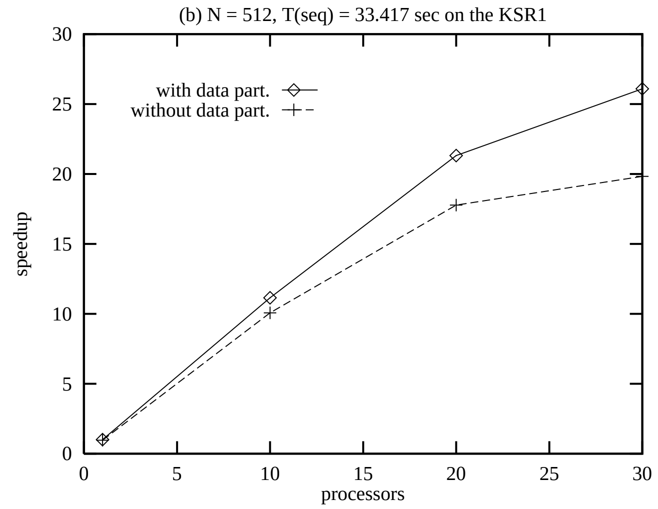


Figure 3.7: Effect of Data Partitioning for Successive Over Relaxation

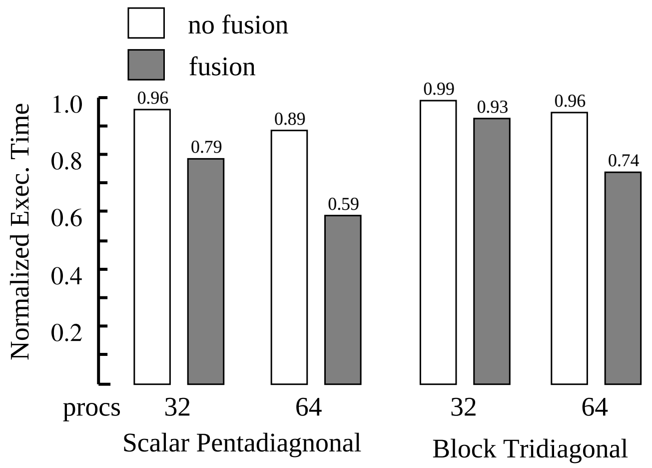


Figure 3.8: Effects of Data Fusion on the NAS Parallel Benchmarks on the KSR1

Two programs, the Scalar Pentadiagonal (SP) and Block Tridiagonal (BT) benchmarks, in the NAS parallel benchmarks require data fusion to effectively use prefetch[2]. Both are CFD applications. In SP, multiple independent systems of non-diagonally dominant, scalar pentadiagonal equations are solved. In BT, multiple independent systems of non-diagonally dominant, block tridiagonal equations are solved. To quantify the effect of fusion in prefetch, we split the fused array into the ones given by the original NAS parallel benchmarks and changed the prefetches for the fused array to prefetches for the split arrays. Figure 3.8 shows the execution times normalized by the execution times without prefetch. Without fusion, prefetch decreases execution time less than 12% in the four cases. With fusion, the reductions are between 5% and 40%. The KSR1 supports at most three outstanding prefetches, but the number of arrays used in the non-fusion version of SP and BT is too large for prefetch to be used on each one. Fusion combines all of them into one array of big records, making prefetch feasible.

3.5 Summary

In this chapter, we have proposed a framework for data transformations and discussed their uses in locality optimization.

- *A General Framework.* In the data transformation framework, we can combine regular data distributions with unimodular transformations to form a rich set of data distributions. The regular data distributions in many parallel language extensions such as HPF are a subset of data transformations.
- *Improving Cache Locality.* Permutation (a unimodular data transformation) can be used to find the best order of array dimensions to exploit cache locality. We have developed a global optimization algorithm to determine whether a permutation is needed for an array in each loop nest. We have applied the optimization algorithm on the NAS kernel benchmarks, and shown that it significantly improves performance and achieves better cache locality than loop transformations do.
- *Reducing Communication.* Since a cache line can hold several words, the number of communication remote accesses can be reduced by mapping individual words in different cache lines into the same cache line. To achieve this, we have defined several data transformations such as partitioning, scaling, and fusion, and shown significant performance benefits from them in several applications.
- *Optimizing Index Computation.* Despite the closed forms for computing indices to access transformed arrays, they involve expensive operations such as modulus, multiplication, and division. Current compilers do not optimize such operations in index computation. We have proposed several optimization techniques that effectively make the overhead of such index computation negligible.

[2] My thanks to Dr. Steve Breit of KSR for providing the KSR version of the NAS parallel benchmarks.

Chapter 4

Locality and Load Balancing

In the previous chapters, we have developed data transformations for improving data locality. For parallel applications with high variance in loop iteration computation time, balancing load among processors is also required for achieving good speedup. Several load balancing schemes have been designed mainly for UMA machines. They handle load imbalance well, but ignore data locality. In this chapter, we address both locality and load balancing issues. We begin by identifying two types of load imbalance, and then propose new load balancing schemes to deal with these two types of load imbalance. We experimentally evaluate these schemes on Hector and the KSR1.

4.1 Load Imbalance

Ideally, each processor should be assigned an equal amount of work so that every processor finishes the work at the same time. But this is difficult to achieve because of:

- **Variance in execution times.** The execution times in many applications are difficult to predict when the code contains some conditional statements or loops for which the amount of computation depends on the input.
- **Variance in partition sizes.** Some problems are difficult to partition into nearly equal-sized sub-problems. The N-body problem and the parallel sorting problem are two examples.

Load imbalance in parallel loops belongs to the first category. When the iteration execution times vary, assigning an equal number of iterations to every processor does not necessarily yield good load balance. Such parallel loops can be found in sparse matrix applications and computations with irregular structures.

Load imbalance in parallel programs based on divide-and-conquer techniques belongs to the second category. After partitioning, the execution times of sub-problems can be accurately estimated. However, partitioning itself usually takes a substantial portion of the total execution time, and the sizes of sub-problems may vary considerablely. Parallel sorting is an example of this type of problem.

The high remote access times on scalable shared-memory multiprocessors further complicate load balancing because data locality must also be taken into account.

4.2 Locality-based Dynamic Loop Scheduling

In this section, we begin with an overview of static and dynamic loop scheduling schemes, then describe a new scheme, Locality-based Dynamic Loop Scheduling (LDS) [1]. Finally, we present experimental results on both Hector and the KSR1.

[1] Joint work with Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik [LTSS93].

4.2.1 Static Scheduling

Static scheduling algorithms, such as block scheduling, cyclic scheduling, and block cyclic scheduling, assign a fixed number of loop iterations to each processor.

Block scheduling divides the loop into blocks of $\lceil N/P \rceil$ iterations, where N is the number of iterations and P is the number of processors. Processor p is assigned the p^{th} block. If the amount of computation performed by each iteration is not the same, then block scheduling can perform poorly because of load imbalance. For example, in an iteration space where the amount of computation per iteration increases linearly, the first few blocks will entail very little computation while the latter ones will involve much more. The first few processors will therefore finish their computations early and have to wait for others to complete, resulting in poor speedup.

Cyclic scheduling assigns loop iterations to processors in a cyclic order, so that processor p will execute the iterations $p, p+P, p+2P, \ldots$, where P is again the number of processors executing the loop. In contrast to block scheduling, cyclic scheduling obtains better load balance for iteration spaces where the amount of computation increases or decreases monotonically through the iterations.

Block-cyclic scheduling is a compromise between block scheduling and cyclic scheduling. This algorithm assigns blocks of a fixed size to processors in a round-robin fashion. If the block size is equal to one, then block-cyclic scheduling degenerates to cyclic scheduling and if the block size is $\lceil N/P \rceil$, then block-cyclic scheduling is the same as block scheduling. Hence, block cyclic scheduling provides a continuum between block and cyclic scheduling algorithms.

The static algorithms ignore the fact that the amount of computation required may differ across iterations, or that it cannot always be determined a priori. (For example, the amount of computation could be dependent on the data.) Moreover, the effective speed of each processor may also differ because of multitasking interference. Therefore static scheduling often suffers from load imbalance, resulting in poor speedup.

4.2.2 Dynamic Scheduling

If the imbalance of computation over iterations becomes large, then it is necessary to dynamically adjust the work assigned to each processor at run-time in order to balance the load. This can be done by grouping one or more iterations to form subtasks which are dynamically allocated and executed by the processors. The subtasks need not be of fixed granularity, and in fact the granularity can vary dynamically. If the granularity is very large ($\lceil N/P \rceil$ iterations per subtask) then we effectively have block scheduling. On the other hand, if the granularity is very small, then the data structure controlling the subtasks could become a bottleneck because of the number of accesses each processor must perform to this structure. Self scheduling, guided self scheduling, factoring, and the trapezoid method are examples of dynamic scheduling algorithms. We describe each of these in turn below.

Self scheduling partitions the loops into subtasks containing one or more iterations [KW85]. Each processor then repeatedly is assigned and executes one subtask at a time until no subtasks are left for processing. If the number of iterations per subtask is fixed and greater than one, then this scheduling strategy is generally referred to as *fixed-size chunking* [KW85].

With fixed-size chunking, it can be difficult to choose the correct granularity. Small granularity increases the overhead of accessing the data structure controlling the remaining subtasks to be executed. Larger granularity can lead to load imbalance when the last set of subtasks is being executed. Hence in the algorithms that follow, the size of the subtasks becomes smaller and smaller with the progress of the computation.

Guided self scheduling(GSS) uses a dynamic subtask granularity of $\lceil n/P \rceil$ iterations, where n is the total number of remaining iterations [PK87]. With this algorithm, the subtasks are composed of a large number of iterations at the start of the computation, then progressively fewer until the size is one. In this scheme, there will be at least $P-1$ subtasks consisting of only one iteration, and each will be executed independently. When the execution time of the iterations differ, it is possible that an early subtask could be so large that it does not complete by the time all other subtasks have completed [HSF92]. This load imbalance problem is addressed by the factoring algorithm.

Factoring is similar to GSS in that the size of the subtask decreases as the computation progresses, but it assigns $\lceil n/(2P) \rceil$ iterations to P consecutive subtasks, where n is equal to the number of remaining iterations

Scheme	No. of Iterations = 500 and P = 4
GSS	125 94 71 53 40 30 22 17 12 9 7 5 4 3 2 2 1 1 1 1
Factoring	63 63 63 63 31 31 31 31 16 16 16 16 8 8 8 8 4 4 4 4 2 2 2 2 1 1 1 1
Trapezoid	62 58 54 50 46 42 38 34 30 26 22 18 14 6
LDS	63 55 48 42 37 32 28 25 22 19 17 14 13 11 10 8 7 7 6 5 4 4 3 3 3 2 2 2 1 1 1 1 1 1 1 1

Table 4.1: Subtask Sizes for Dynamic Scheduling Algorithms

at the beginning of these allocations [HSF92]. Hence P consecutive subtasks will be of the same size, before the granularity is decreased. If the variance of the amount of computation performed by each iteration is large, then factoring performs better than GSS [HSF92].

The *Trapezoid* method also assigns a decreasing number of iterations to subtasks and thus is a variation of GSS. In this case, however, the subtask size decreases linearly instead of exponentially [TN91]. The total number of iterations, N, is partitioned into $S = \lceil 2N/(f+1) \rceil$ subtasks, where $f = \lfloor N/(2P) \rfloor$ is the size of the first task. Consecutive subtasks differ in size by $\lfloor (f-1)/(S-1) \rfloor$ iterations.

For comparison, the subtask sizes employed by the dynamic scheduling algorithms GSS, factoring, and the trapezoid method for a problem size with 500 iterations executing on four processors are given in Table 4.1.

In general, dynamic scheduling algorithms can achieve good load balance, but at the cost of decreased locality in data accesses, since each subtask may be scheduled on any of the processors regardless of the location of the data it must access. The cost of an average memory access would increase on systems with non-uniform memory access cost. This can lead to a decrease in performance, not only because of increased latencies to access the remote data, but also because of increased network traffic and congestion.

4.2.3 Locality-based Dynamic Scheduling

We propose a new scheduling algorithm, Locality-based Dynamic Scheduling (LDS), that addresses both locality and load balancing. LDS (see Figure 4.1) is based on the following principles:

1. The data space is partitioned to reside in the local memories associated with the P processors. Typical data partitions are block, cyclic, and block-cyclic. Often, the partitioning strategy chosen is dictated by the rest of the computation.

2. Each processor, when it is ready to execute the next subtask, computes the size of the subtask. The size can be chosen as in any of the dynamic scheduling algorithms as a function of the number of remaining iterations and the number of processors. In our experiments, like Factoring, we set the subtask size to $S = \lceil n/(2P) \rceil$ (where n is the number of remaining unscheduled iterations and P is the number of processors). This creates an initial subtask about half as large as that of GSS in order to avoid an overly large initial subtask size (see Table 4.1).

3. Once the subtask size has been determined, it must be decided which iterations to execute as part of the subtask. The dynamic scheduling algorithms sequentially take iterations from the loop iteration space; that is, the first subtask of size $S1$ includes iterations $1, 2, \ldots, S1$, the second subtask of size $S2$ includes iterations $S1+1, \ldots, S1+S2$, and so on. In LDS, on the other hand, the iterations are chosen such that locality is maximized. For example, if the data distribution is cyclic, and processor p has to execute a subtask of $S1$ iterations, then it executes the iterations $p+P, p+2P, \ldots, p+P*S1$. If the data distribution is block then the subtask would include iterations $p*B+1, p*B+2, \ldots, p*B+S1$, where B is the block size.

4. If all the scheduled local iterations are completed, $\lfloor S/R \rfloor$ iterations are acquired from some processor that has more than the current average number of unscheduled iterations remaining. The parameter R represents the ratio of the average local iteration execution time to the average remote iteration

1. Partition iterations among processors based on data distribution.
2. While unscheduled iterations remains, processor p should:
   (a) Determine the subtask size $S = \lceil n/(2P) \rceil$ based on the total number of unscheduled iterations (n) and the number of processors (P).
   (b) Get iterations for the subtask.
      - If it has $r > 0$ locally assigned, unscheduled iterations, then the subtask includes $\min(r, S)$ of those iterations.
      - Otherwise, it searches processors $p+1, \ldots, P, 1, \ldots, p-1$, and acquires $\lfloor S/R \rfloor$ iterations from the first processor with a number of unscheduled iterations greater than the average ($2S$). The parameter R represents the slowdown factor for remote iterations.
   (c) Execute the subtask.

   end while

Figure 4.1: Locality-based Dynamic Scheduling Algorithm

execution time. It is dependent on both the imbalance in the application and the remote memory access time. To simplify the scheduling algorithm, we choose $R = 2$ for Hector and $R = 4$ for the KSR1, based on the remote memory access times. The value of R indicates the tradeoff between load imbalance and locality. For multiprocessors with very high remote memory access times (*i.e.*, a very large R), dynamic scheduling is prohibitively costly, and thus LDS degenerates to a static loop scheduling algorithm.

The LDS algorithm is related to the *affinity scheduling* algorithm (AFS) proposed by Markatos and LeBlanc in that AFS also takes locality into account [ML94]. AFS divides the iterations of a loop into block partitions of $\lceil N/P \rceil$ iterations, where N is the total number of iterations and P is the number of processors, and assigns each partition to a different processor. When a processor becomes free, it takes the next subtask, which consists of one kth of the number of iterations from its local partition, where k is a parameter of the algorithm that is chosen statically between 2 and P. Once the entire local partition has been executed, the processor takes a fraction $\lceil 1/P \rceil$ of iterations from the processor with the most remaining iterations.

LDS is different from AFS in the following ways:

- AFS assumes that data accessed in iteration j is likely to be adjacent to the data used in iteration $j+1$, and thus partitions the iteration space into blocks and assigns a block to each processor. Therefore, memory locality can be exploited only when the data is also partitioned and distributed in blocks. LDS, on the other hand, takes data placement into account, by always having the processor first execute those iterations which have the data local to the processor. For this reason, LDS can accommodate other data partitioning methods, such as cyclic or block-cyclic.

- In AFS, each processor independently schedules iterations from its local partition using a parameter k. The best value for k will depend on the application and may be difficult to choose. If k is small, then a processor with an exceptionally large proportion of the workload assigned to it could make the size of the first subtask too large so that later dynamic scheduling will not be able to adjust for the load imbalance. The maximum load imbalance in a loop with a linear iteration space in AFS is $\frac{N(P-k)}{P(P-1)k} + 1$ iterations. When k approaches P, on the other hand, the worst-case load-imbalance approaches that of GSS, but at the cost of an increase in the number of synchronization operations by a factor of P, since the total number of lock operations performed by AFS is $O(kP \log \frac{N}{kP}) + O(P \log \frac{N}{P^2})$ [ML94]. In LDS, the worst-case load-imbalance will be one iteration and the number of synchronization operations will be $O(P \log(N))$.

	Do not consider Locality	Consider Locality
Static	Block Cyclic Block-cyclic	Block-D Cyclic-D Block-cyclic-D
Dynamic	GSS Self Factoring Trapezoid	AFS LDS

Table 4.2: Comparing Various Loop Scheduling Algorithms

	AFS	LDS
Locality	block only	any data distritions
Lock Operations	$O(kP\log\frac{N}{kP})+$ $O(P\log\frac{N}{2})$	$O(P\log(N))$
Max Imbalance (in iterations)	$\frac{N(P-k)}{P(P-1)k}+1$	1
Search for Max Loaded	yes	no

Table 4.3: Comparison of AFS and LDS

- In AFS, when a processor finishes its local partition, it searches for the processor with the most unscheduled iterations. Such a search requires P remote memory accesses. In LDS, since the current average number of unscheduled iterations is $2S$, overloaded processors can be identified by comparing their numbers of unscheduled iterations with $2S$. The extensive search is thus avoided.

Table 4.2 gives a comparative classification of the scheduling algorithms we have discussed. In the loop scheduling schemes of Block-D, Cyclic-D, and Block-cyclic-D, data is distributed in a coordinated fashion to obtain data locality. The differences between AFS and LDS in particular are summarized in Table 4.3. Differences in performance are shown in the next section.

4.2.4 Experimental Results

We present performance results from experiments involving four benchmark programs: Matrix Multiplication (MM), LU Decomposition (LU), Red-Black Successive Over Relaxation (SOR), and Conjugate Gradient (CG).

- Three programs (MM, LU, and SOR, described in Chapter 2) do not have load imbalance in their parallel loops. The array size for MM and LU is 400×400. SOR contains 400×400 unknowns stored in a two-dimensional array.

- Conjugate Gradient (CG) is a sparse linear system solver from the NAS parallel benchmarks. The main data structures are a sparse matrix and several vectors. The sparse matrix-vector multiplier routine dominates the total execution time. Load imbalance is thus dependent on the characteristics of the sparse matrix.

Figures 4.2-4.5 show the speedups of these applications listed above when run with the different scheduling policies.

In matrix multiplication, all processes must access all of the matrix B. Assuming B does not fit in the cache, then the overhead of accessing B will dominate the total cost of accessing the data elements. Good

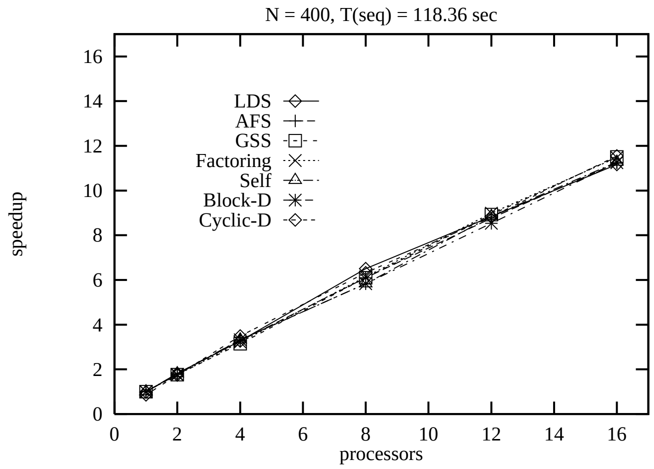


Figure 4.2: Speedups for Matrix Multiply on Hector

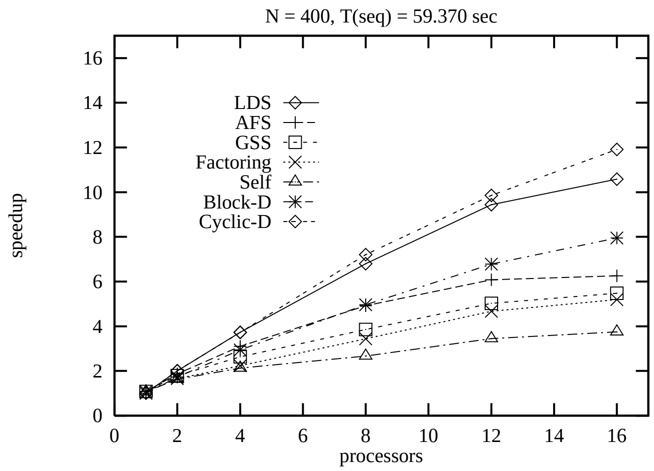


Figure 4.3: Speedups for LU Decomposition on Hector

locality in accessing the elements of matrices A and C is automatically achieved through caching. For this reason, and because the load in this computation is well balanced, all of the scheduling algorithms perform equally well, as shown in Figure 4.2.

Our results for matrix multiplication differ from those of a similar experiment performed on the RP3 by Hummel *et al.* [HSF92]. In our experiments, static partitioning, namely Cyclic-D, performs marginally better than the other scheduling algorithms. The results of Hummel *et al.* indicate that static partitioning performs far worse than the dynamic schemes. We believe this discrepancy is due to a mismatch between the data partitioning and loop partitioning in the RP3 experiments, making the static algorithm perform poorly.

For LU decomposition, static cyclic scheduling (Cyclic-D) outperforms all other scheduling algorithms, because cyclic loop scheduling matches the cyclic data partitioning, and balances the load well for the iteration spaces of parallel loops reducing with the outermost loop. AFS does not perform well either because its initial block partitioning sacrifices some data locality. [2] LDS performs almost as well as Cyclic-

[2] AFS can be modified to accommodate different regular data distributions.

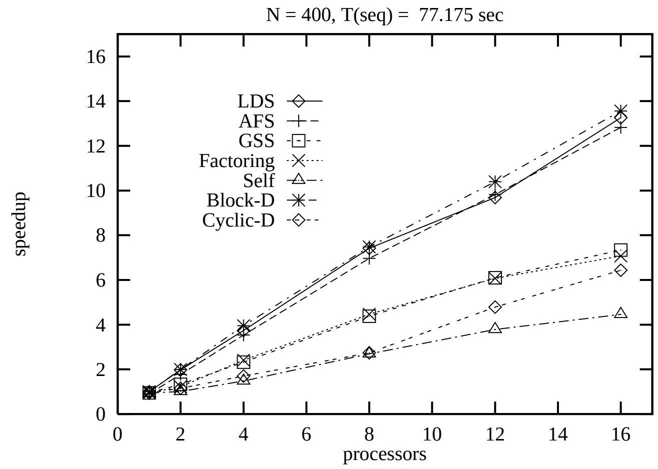


Figure 4.4: Speedups for Successive Over Relaxation on Hector

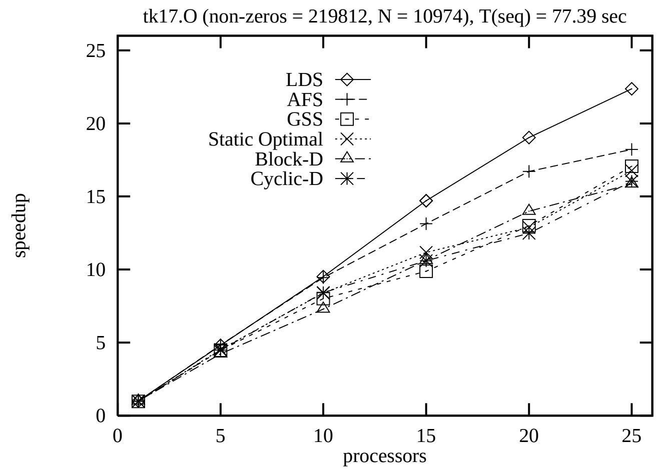


Figure 4.5: Speedups for Conjugate Gradient on the KSR1

D, and better than the other algorithms for the same reasons. The speedup of the program using LDS is slightly lower than that for Cyclic-D because of the run-time scheduling overhead. The results are shown in Figure 4.3.

For SOR, the static scheduling algorithm that matches the data partitioning, namely Block-D, performs best, and the best results are obtained when the data and iteration spaces are partitioned into blocks. Again, LDS performs almost as well as Block-D because its iteration space is effectively also partitioned into blocks, given the block distribution of data. In this case, affinity scheduling (AFS) performs equally well because it also uses block partitioning. From Figure 4.4, it is interesting to note that the performance of the other dynamic algorithms is substantially worse than Block-D, LDS, and AFS because of the lack of data locality.

In Conjugate Gradient, the sparse matrix (as specified in the NAS parallel benchmarks) has non-zero elements spread uniformly through the array, so no load imbalance exists. We instead use the matrix "`tk17.O`", one of the standard benchmark sparse matrices from Boeing. The parallel execution times of the matrix-vector multiplier in CG with different scheduling schemes on the KSR1 are shown in Figure 4.5. Table 4.4 lists the improvements by LDS over other schemes and the reasons for the differences. Despite retaining data locality, Block-D suffers from load imbalance and performs the worst. Cyclic-D improves

	LDS Improvement (%)	Imbalance	Poor locality	Scheduling overhead
Block-D	41	yes		
Cyclic	40	yes	yes	
GSS	31		yes	yes
AFS	22			yes
Static Opt.	33	yes (?)		

Table 4.4: The Improvements by LDS over Other Algorithms in Conjugate Gradient on the KSR1

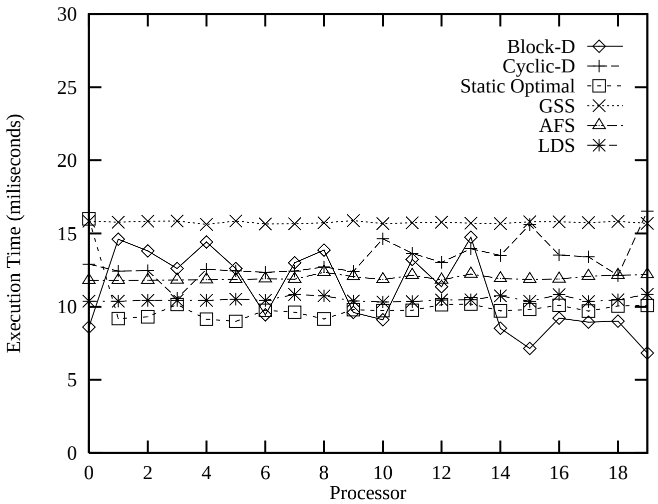


Figure 4.6: Load Imbalance in Conjugate Gradient for 20 Processors on the KSR1

load balancing at the cost of loss of data locality. GSS also ignores data locality, but it balances load well, and thus slightly outperforms Cyclic-D. AFS attains both data locality and load balance, so it outperforms other schemes except LDS. AFS is slower than LDS because, when a processor needs to get iterations from other processors, AFS must search for the most loaded processor, but in LDS, because of a global dynamic scheduling scheme, we can determine whether a processor is relatively overloaded without searching.

In experimenting with CG, we also include a "Static-Optimal" scheme. Because the sparse matrix is read-only and CG involves many iterations [3], we can partition the computation evenly among processors by counting non-zero elements in rows. This scheme is called "Static-Optimal" because it retains data locality as well. We expected that Static-Optimal should outperform all other schemes. However, the experiments give a counter intuitive result: Static-Optimal performs much worse than either AFS or LDS.

Figure 4.6 explains the reason for Static-Optimal's poor performance. It shows the execution time on each processor when CG is executed on 20 processors. In Static-Optimal, despite the equal amount of computation on every processor, the execution times still vary among processors, with a heavy load on processor 0. This is because:

- Processor 0 is responsible for more rows than others because the first rows contain fewer non-zero elements. This causes more remote memory accesses on processor 0, and thus slows down processor 0 significantly.
- The unpredictable cache behavior in sparse matrices also contributes to the variance of execution time.

Three scheduling schemes, GSS, AFS, and LDS, maintain load balance well, but LDS achieves the best execution time among the three. In AFS, a processor that has finished its local partition searches for the

[3] The matrix-vector multiplication executes 375 times

most loaded processor in order to do load balancing. In the run on 20 processors, the average number of searches for AFS is 3.14 per processor, which accounts for the 15% difference in performance between AFS and LDS.

In summary, we have shown that data locality is an important factor to consider in loop scheduling. While most existing dynamic scheduling algorithms manage load imbalances well, they fail to take locality into account and therefore perform poorly on parallel systems with non-uniform memory access times. We have presented a new scheduling algorithm, LDS, that is dynamic, yet takes locality into account. We have presented experimental results that indicate:

- Scheduling algorithms do not perform well on scalable multiprocessors over a variety of applications if they do not take locality into account.
- Static scheduling algorithms are suitable for applications with low variance in loop iteration execution times.
- Dynamic scheduling is necessary for applications in which subtask execution times are highly variable and unpredictable.
- LDS achieves both locality and load balance with low overhead, and is thus competitive to the other dynamic scheduling schemes we investigated.

4.3 Overpartitioning

In this section, we study the second category of load imbalance, which is due to high variability in partition size. We first propose a new scheme called *overpartitioning*, then apply it to two parallel sorting algorithms. In Section 4.3.4, we present experimental results.

4.3.1 The Overpartitioning Technique

In some problems, it is inherently difficult to partition the data into nearly equal-sized sub-problems. In parallel sorting, an unsorted list is first partitioned into P sublists by $P-1$ pivots, and then each processor sorts one of the sublists independently. The final sorted list is the concatenation of these sorted sublists. The partitioning techniques in parallel sorting algorithms that have been proposed assume that the number of partitions should be equal to the number of processors. This assumption makes dynamic load balancing impossible. We propose a new technique, called *overpartitioning*, which:

- allows the number of partitions to be larger than the number of processors,
- tolerates variability in sub-problem size, and
- achieves good load balance by dynamic scheduling.

A canonical form of the overpartitioning technique involves three phases:

1. Partition a problem into kP sub-problems, where k is called the *overpartitioning ratio*.
2. Build a task queue. Each sub-problem corresponds to a task, and the tasks are placed in a task queue in decreasing order of sub-problem size. [4]
3. Dynamically schedule the tasks. Each processor gets one task from the task queue at a time, and processes the corresponding sub-problem, until the task queue is empty.

Overpartitioning may also be applied to other problems that can be partitioned into relatively independent sub-problems and the sub-problem size can be quickly determined. In the rest of the section, we incorporate overpartitioning into parallel sorts and present experimental results.

[4] Finding an optimal solution to the scheduling problem is equivalent to the bin-packing problem (NP-complete). With the largest-job-first policy, the approximate solution is at most 11/9 times worse than the optimal solution (see page 127 in [GJ79]).

	Single-step	Multi-step
merge-based	quickmerge [Qui89] PSRS [SS92, LLS+93]	Batcher's bitonic sort [Bat68] parallel merge sort [BH85, Col88] smoothsort [Pla89] Nassimi and Sahni's sort [NS82] column sort [Lei85] Tridgell and Brent's sort [TB93] snakesort [BR93]
quicksort-based	parallel quicksort [Ble90, LT90, FP92] parallel sample sort [HC83, BLM+91]	flashsort [RV87] B-flashsort [HPR92] hyperquicksort [Wag90, Qui89] Kalé and Krishnan sort [KK93]

Table 4.5: Classification of Parallel Sorting Algorithms

4.3.2 Locality and Load Balancing in Parallel Sorting

In a typical parallel sorting algorithm, each processor's memory contains a portion of the list of n elements to be sorted. In a merge-based approach, each processor sorts the portion it holds initially by some sequential sorting algorithm, and then all the sorted portions are exchanged among all the processors and merged. Alternatively, in a quicksort-based approach, the unsorted list is partitioned into a number of progressively smaller sublists defined by selected pivots. Then, the processors employ a sequential sorting algorithm to sort the sublists for which they are responsible. The merging of the sorted portions of different processors in the merge-based approach and the partitioning into sublists in the quicksort-based approach can be achieved either in one step or many steps.[5] Existing parallel sorting algorithms can therefore be classified as *single-step* or *multi-step* (as in Table 4.3.2).

The performance of a parallel sorting algorithm is determined by the communication cost and the degree of load balancing. Single-step algorithms incur low communication cost, because they move each element at most once. However, single-step algorithms often have poor load balancing because it is difficult to derive nearly equal sized sublists in the absence of global information and single-step algorithms cannot refine the sublists once they are derived. On the other hand, multi-step algorithms typically incur higher communication cost because elements may be moved several times between processors.

Oversampling and *regular sampling* techniques have been developed for reducing load imbalance in single-step algorithms.

- *Parallel Sorting by Regular Sampling* (PSRS) [SS92, LLS+93]

  As a merge-based single-step algorithm, it sorts the portions, and then uses *regular sampling* to select pivots as follows: It picks $P-1$ pivots by first choosing P equally spaced candidates from each portion, and then selecting $P-1$ equally spaced pivots from the sorted P^2 candidates. The regular sampling guarantees to balance the load among processors within a factor of two of optimal in theory, and within a few percent of optimal in practice. However, regular sampling requires that the local portions be sorted, so it cannot be applied to quicksort-based algorithms.

- *Parallel Sample Sort* (PSS) [HC83, BLM+91]

  As a quicksort-based single-step algorithm, it uses oversampling in selecting pivots: it picks $P-1$ pivots by randomly choosing sP candidates from the entire list ℓ, where s is the *oversampling ratio*, and then selecting $P-1$ equally spaced pivots from the sorted candidates. A larger oversampling ratio results in better load balancing, but increases the cost of selecting pivots.

[5] A single-step algorithm will merge the portions in all the processors or partition the sublists and assign them to all the processors in one step.

4.3.3 Parallel Sorting by Overpartitioning (PSOP)

Overpartitioning can be used to do parallel sorting, and the resulting algorithm is called *Parallel Sorting by Overpartitioning* (PSOP). To sort a given list ℓ of n unsorted keys on P processors, PSOP proceeds in five phases:

1. Initially, processor i has ℓ_i, a portion of size n/P of the unsorted list ℓ in its memory.
2. *Selecting Pivots.* A sample of skP candidates are randomly picked from the list, where s is the oversampling ratio, and k is the overpartitioning ratio. Each processor picks sk candidates and passes them to a designated processor. These candidates are sorted, and then $Pk-1$ pivots are selected by taking the $s^{th}, 2s^{th}, \ldots, (kP-1)s^{th}$ candidates from the sample. The selected pivots, $d_1, d_2, \ldots, d_{kP-1}$ are made available to all the processors.
3. *Partitioning.* On each processor i, ℓ_i is divided into pieces $\langle \ell_{i1}, \ldots, \ell_{i(kP)} \rangle$ using the $kP-1$ pivots, so that ℓ_{ij} is the set of elements on processor i that are between pivots $j-1$ and j. Since the pivots have been sorted, binary partitioning is applied as follows: processor i first uses the middle pivot, $d_{\lfloor (kP-1)/2 \rfloor}$, to partition ℓ_i into ℓ_i^{left} and ℓ_i^{right}, then recursively uses the left pivots to partition ℓ_i^{left} and the right pivots to partition ℓ_i^{right} until all ℓ_{ij} are identified. A sublist S_j is the union of ℓ_{ij} with i ranging over all processors:

   $$S_j = \cup_{i=1}^{P} \ell_{ij}$$

4. *Building a task queue.* Let $T(S_j)$ denote the task of sorting S_j. The size of each sublist can be computed :

   $$|S_j| = \sum_{i=1}^{P} |\ell_{ij}|$$

   Also, the starting position of sublist S_j in the final sorted array can be calculated:

   $$\sigma_j = 1 + \sum_{h=1}^{j-1} |S_h|$$

   A task queue is built with the tasks ordered from largest sublist size to smallest.
5. *Scheduling tasks and sorting sublists.* Each processor repeatedly takes one task, $T(S_j)$, at a time from the head of the queue. It processes the task by (a) copying the P parts of the sublist, $\langle \ell_{1j}, \ldots, \ell_{Pj} \rangle$, into the final array at position σ_j to $\sigma_j + |S_j| - 1$, and (b) applying a sequential sort to the elements in that range. This process continues until the task queue is empty.

Figure 4.7 illustrates the PSOP approach. Since the tasks in the queue are in decreasing order of sublist size, in Phase 5, every processor gets larger tasks at the beginning, and smaller tasks towards the end. Overpartitioning leads to a sufficient number of tasks of varying size so that PSOP achieves good load balancing. PSOP also minimizes communication cost because each element moves between processors at most once.

The five phases in PSOP define a general framework for parallel sorting algorithms with overpartitioning, where any sequential sorting algorithm can be used by each processor in the last phase.

4.3.4 Load Balancing in PSOP

There are two parameters in PSOP that influence the degree of load balancing, namely, the oversampling ratio and the overpartitioning ratio. The oversampling ratio governs the variance of sublist sizes. A larger oversampling ratio produces sublists with a smaller variability in size. The overpartitioning ratio determines the total number of sublists and the average sublist size. A larger overpartitioning ratio gives more sublists but with smaller average and maximum sublist sizes, leading to better load balancing in Phase 4. However, the time spent in Phase 1 increases with the oversampling ratio and the overpartitioning ratio. Therefore, it is necessary to choose appropriate values for the two parameters in order to achieve good performance.

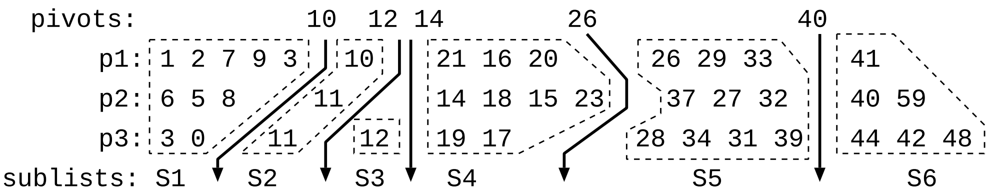
Phase 1. Initially, a list to be sorted is distributed among processors $(P = 3, n = 39)$.

p_1 :	10	21	1	2	26	29	7	33	9	20	3	16	41
p_2 :	6	40	59	5	14	37	18	8	27	15	32	23	11
p_3 :	11	44	12	3	19	28	34	31	42	17	48	0	9

Phase 2. Randomly pick up skP candidates $(s = 1, k = 2)$, sort them, and select $(kP - 1)$ from them as pivots.

skP candidates :	10	26	40	14	44	12
sorted :	10	12	14	26	40	44
$(kP - 1)$ pivots :	10	12	14	26	40	

Phase 3. Each processor uses the pivots to partition its own portion locally.



Phase 4. Compute the sublist sizes and their final positions in the sorted list and build a task queue.

sublists :	S_1	S_2	S_3	S_4	S_5	S_6
sizes :	10	3	1	9	10	6
final start positions :	0	10	13	14	23	33

task queue : $T(S_1), T(S_5), T(S_4), T(S_6), T(S_2), T(S_3)$

Phase 5. When the queue is not yet empty and a processor is available, it gets a task from the queue, copies the pieces of the sublist to the target array, and sorts the sublist.

Figure 4.7: An Example Using PSOP

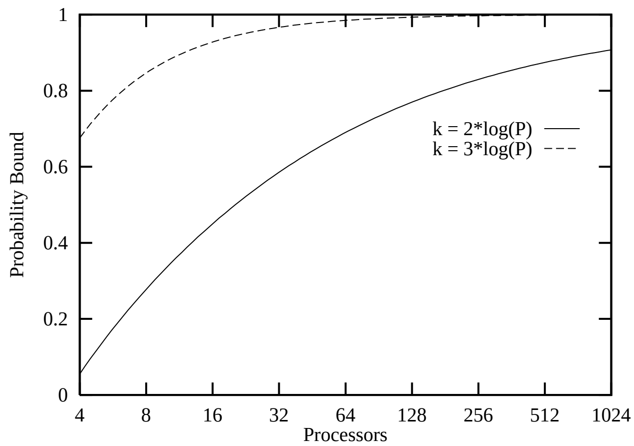


Figure 4.8: Lower Bounds on $Pr[\max_{1 \le j \le kP} |S_j| \le n/P]$

Maximum Sublist Size

Blelloch et al. [BLM+91] have shown that the sublist sizes can be reduced by increasing the oversampling ratio. They have proved that for $(P-1)$ pivots and an oversampling ratio s, the probability that the maximum sublist size is greater than $\alpha(n/P)$ $(\alpha > 1)$ is

$$Pr[\max_{1 \le i \le P} |S_i| > \alpha(n/P)] \le ne^{-(1-1/\alpha)^2 \alpha s/2}$$

We establish the relationship between the maximum sublist size and the overpartitioning ratio as follows: **Theorem:** *For an unsorted list of size n, $(kP-1)$ pivots (with $k \ge 2$) partition the list into kP sublists such that the size of the maximum sublist is less than or equal to n/P with probability at least*

$$Pr[\max_{1 \le j \le kP} |S_j| \le \frac{n}{p}] \ge 1 - 2P(1 - \frac{1}{2P})^{kP-1}$$

. **Proof:** The final sorted list can be viewed as $2P$ segments of size $\frac{n}{2P}$. If every segment contains at least one pivot, then $\max_{1 \le j \le kP} |S_j| \le \frac{n}{P}$, no matter where the pivots are located within their respective segments. Consider one of the segments. Since the pivots are chosen randomly, the probability that a specific pivot is not in the segment is $(1 - \frac{1}{2P})$. Since the $q = kP - 1$ pivots are selected independently, the probability that none of the pivots are in the segment is

$$(1 - \frac{1}{2P})^q$$

Therefore, the probability of at least one of the $2P$ segments not containing any pivots cannot exceed

$$2P(1 - \frac{1}{2P})^q$$

In other words, every segment contains at least one pivot with probability at least

$$1 - 2P(1 - \frac{1}{2P})^q$$

□

Figure 4.8 shows the lower bounds on the probabilities that the maximum sublist size is less than n/P for overpartitioning ratios of $k = 2\log_2(P)$ and $k = 3\log_2(P)$. When k grows in proportion to $\log_2(P)$, the lower bound on probability approaches one as P increases.

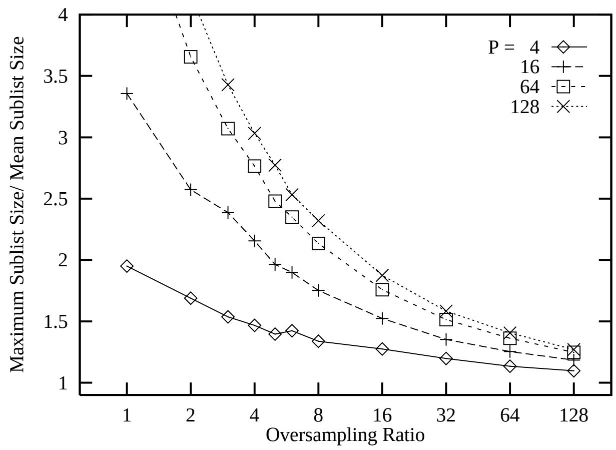


Figure 4.9: Effect of Oversampling on Load Balancing

Tradeoff between s and k

To illustrate the effects of the overpartitioning ratio k and the oversampling ratio s on load balancing, we performed 100 tests for each combination of the parameters s, k, and P. For each test, we randomly generated one million ($n = 2^{20}$) 32-bit integers and used the combination of parameters to perform the partitioning. These integers were uniformly distributed in $[0, 2^{31} - 1]$. The results of this experiment are shown in Figures 4.9 and 4.10.

Oversampling. Figure 4.9 shows the effect of oversampling on the sublist sizes.

- The ratio of maximum sublist size to mean sublist size decreases as s increases. However, for $P \geq 64$, it is still around 1.3 times the mean sublist size even when s is as high as 128. Thus, oversampling can reduce load imbalance only to a certain degree.
- The ratio of maximum sublist size to mean sublist size increases as the number of processors increases.
- Small values of s make the maximum sublist size sufficiently small for overpartitioning.

Overpartitioning. The effect of overpartitioning on load balancing is shown in Figure 4.10 where the oversampling ratio s was set to 3 so that the maximum sublist size is only a few times larger than the average sublist size.

- In Figure 4.10 (a), load imbalance decreases dramatically as k increases. For P from 4 to 128, the average value of load imbalance is only about 5% when k is as small as 5.
- Figure 4.10 (b) shows that load imbalance also depends on the number of processors P. When $k = \log_2(P)$, the load imbalance decreases slightly with P. This is consistent with the Theorem and suggests that $k = \log_2(P)$ is suitable for balancing the load.

4.3.5 PSOP with Two Sequential Sorts

In this section, we specify and analyze two versions of PSOP that use a comparison-based sort (quicksort) and a non-comparison based sort (radix sort) respectively. The resulting algorithms are called *Parallel Quicksort by Overpartitioning (PQOP)* and *Parallel Radix sort by Overpartitioning (PROP)* respectively.

Table 4.6 lists the parameters involved in the complexity analysis of these algorithms. In shared-memory multiprocessors with non-uniform memory access time, remote memory accesses must be distinguished from local memory accesses in doing performance analysis. The value of Q is the ratio of the total execution time for quicksort to the number of comparisons (in CPU cycles). The value of t_r, the mean time to access remote

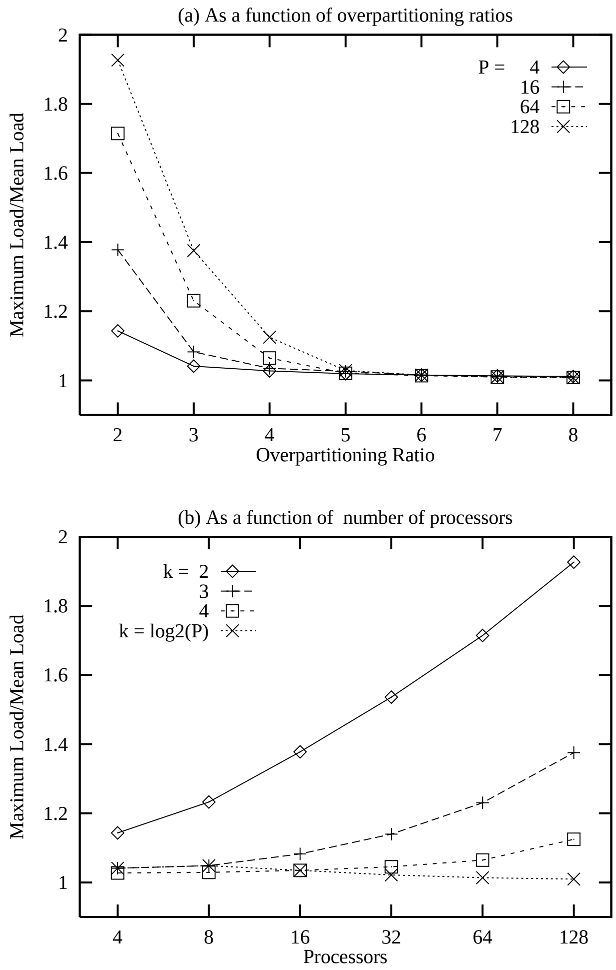


Figure 4.10: Effect of Overpartitioning on Load Balancing

	Meaning
n	number of integers to be sorted
P	number of processors
C	constant factor in quicksort
Q	unit cost in quicksort
R_b	unit cost in b-bit radix sort
W	number of integers per cache line
t_r	mean time to access remote memory

Table 4.6: Parameters for Analysis of PQOP and PROP

memory, depends not only on how many processors are allocated, but also on which of the processors are used.

Parallel Quicksort by Overpartitioning (PQOP)

In PQOP, we choose pivots in Phase 2 just as in sequential quicksort, and use binary partitioning in Phase 3. The last phase of PSOP uses a sequential quicksort to sort the sublists.

Selecting Pivots. Each processor randomly selects sk candidates and puts them into an array, requiring $t_r sk/W$ time for remote memory access. One of the processors then sorts the candidates by quicksort and determines the $(kP-1)$ pivots.[6] It takes $t_r skP/W$ time to access the skP candidates from other processors and $CQskP\log_2(skP)$ time to sort them. The time for selecting pivots is therefore:

$$T_{pivot} = CQskP\log_2(skP) + t_r\frac{sk(P+1)}{W}$$

Partitioning. All processors except one need to read the $kP-1$ pivots remotely in about $(t_r kP/W)$ time. Since a recursive binary partitioning is used, the depth of the recursion is $\log_2(kP)$, resulting in the partitioning time $Q\frac{n}{P}\log_2(kP)$. So the time for this phase is:

$$T_{part} = t_r\frac{kP}{W} + Q\frac{n}{P}\log_2(kP)$$

Building the Task Queue. Since each sublist is formed by the union of the corresponding parts on all processors, computing the size of a sublist requires P remote memory accesses and P operations. Each processor needs to compute k sublist sizes independently in time kP with remote memory accesses in time $t_r kP/W$. The task queue is built by sorting the tasks in decreasing sublist sizes.[6] The time to construct the task queue is therefore:

$$T_{queue} = t_r\frac{k(P+1)}{W} + kP + CQkP\log_2(kP)$$

Scheduling Local Sorting. Each processor gets a task from the task queue and copies the sublist of the task into the destination array at its final location. Thus, the total time for remote memory copying on all the processors is $t_r n/W$. To find the total time of sorting all the sublists, denoted by T_{sort}^{all}, we observe that the partitioning in Phase 2 and the sorting in Phase 4 perform the work equivalent to quicksort, thus

$$T_{sort}^{all} = CQn\log_2(n) - Qn\log_2(kP)$$

By choosing appropriate values for k and s to achieve good load balancing, the time in this phase on every processor should be:

$$T_{sched} = (t_r\frac{n}{W} + T_{sort}^{all})/P$$

Therefore, the total execution time of PQOP is:

$$T_{PQOP} = T_{pivot} + T_{queue} + T_{part} + T_{sched}$$

or

$$T_{PQOP} = \frac{CQn\log_2(n)}{P} + CQskP\log_2(skP) + CQkP\log_2(kP) + \frac{2t_r skP}{W} + \frac{t_r n}{PW}$$

[6] In our current implementation, one processor is used to do this sorting, but this phase could be parallelized when P is large.

Phase	Complexity in PQOP	Complexity in PROP
T_{pivot}	$CQskP\log_2(skP) + t_r sk(P+1)/W$	
T_{part}	$t_r kP/W + Q(n/P)\log_2(kP)$	$R_H(n/P)$
T_{queue}	$t_r k(P+1)/W + kP + CQkP\log_2(kP)$	$t_r k(P+1)/W + kP + CQ(H-1)2^{H-1}$
T_{sort}^{all}	$CQn\log_2(n) - Qn\log_2(kP)$	$R_B\lceil(32-H)/B\rceil n$
T_{sched}	$(t_r n/W + T_{sort}^{all})/P$	$(t_r n/W + T_{sort}^{all})/P$

Table 4.7: Complexity Analysis of PQOP and PROP

Parallel Radix Sort by Overpartitioning (PROP)

We assume that the list to be sorted contains 32-bit positive integers. In PROP, we use a radix partitioning in Phase 3 and radix sort in Phase 4. Phase 2 is not needed because the radix partitioning is based on the first H-bits.

Radix Partitioning. Each processor performs radix partitioning independently on a portion of the list as follows:

1. Build a histogram of the first H-bits of the locally held elements;
2. Pass over the keys and move each one to its appropriate position in a second array as determined by the histogram counts within the processor.

The time to do radix partitioning is therefore:

$$T_{part} = R_H \frac{n}{P}$$

Building a Task Queue. Each value of the first H-bits defines a sublist that corresponds to a task, so the total number of tasks is 2^{H-1}.[7] The size of the sublist is the number of occurrences of each histogram entry over all the processors. These tasks are sorted by quicksort in decreasing order of the sublist size in time $CQ2^{H-1}\log_2(2^{H-1})$, or $CQ(H-1)2^{H-1}$. So, the complexity of this phase is similar to that in PQOP.

$$T_{queue} = t_r \frac{k(P+1)}{W} + kP + CQ(H-1)2^{H-1}$$

Scheduling Local Sorting. This component is similar to that in PQOP except that radix sort is applied to sort each sublist. Since all the numbers in the same sublist have the same value in their first H-bits, radix sort is applied only to the last $(32-H)$ bits. If each pass in the radix sort performs on B bits, then the number of passes should be $\lceil \frac{32-H}{B} \rceil$. The time for this phase is thus:

$$T_{sched} = t_r \frac{n}{PW} + R_B \lceil \frac{32-H}{B} \rceil \frac{n}{P}$$

Therefore, the total time for PROP is:

$$T_{PROP} = T_{part} + T_{queue} + T_{sched}$$

or

$$T_{PROP} = R_H \frac{n}{P} + R_B \lceil \frac{32-H}{B} \rceil \frac{n}{P} + CQ(H-1)2^{H-1} + \frac{2t_r kP}{W} + \frac{t_r n}{PW}$$

Since radix sort has a random access pattern, sorting a sublist of size larger than the cache size has a low cache hit ratio, thus degrading the performance. We thus choose a value for H such that most sublists have size smaller than the cache size.

Table 4.7 summarizes the complexity analysis of PQOP and PROP.

[7] Positive integers are used, so the first bit is always zero.

	KSR1	Hector
C	1.32	1.32
Q	18.06 (cycles)	18.78 cycles
W	32	4
t_r	depends on P	depends on P

Table 4.8: Parameter Values for Hector and the KSR1

n	100K	256K	512K	800K	1M	8M
KSR1	2.05	5.60	12.00	19.51	25.46	234.26
Hector	2.70	7.34	16.40	25.67		

Table 4.9: Sequential Execution Times of Quicksort (in seconds)

4.3.6 Experimental Results

The PQOP and PROP algorithms have been implemented on both the KSR1 and Hector multiprocessors. The measured timing does not include the times for initializing the list and confirming the correct order of the final list. The cost of handling page faults is not included in the timing because the current KSR OS does not handle page faults efficiently. We believe that this cost will be negligible in an improved operating system.

The complete C code of PQOP for the KSR1 is given in Appendix B.

Performance of PQOP

Speedup. The speedups are calculated relative to the execution times of sequential quicksort given in Table 4.9. When the data size is too large to fit in the local memory, we use $CQn\log_2(n)$ to estimate the sequential execution time, where C is the constant factor in quicksort, and Q is the unit cost in quicksort. The value for C is 1.32 in our implementation of quicksort in sorting uniformly distributed integers. The average value of Q was empirically determined to be 18.06 CPU cycles on the KSR1 and 18.78 cycles on Hector.

Figure 4.11 (a) shows speedups on the KSR1. The speedup for sorting 100 K integers gradually stops increasing, resulting in an execution time of 69 milliseconds and 30-fold speedup on 64 processors. When a larger number of elements are sorted, the speedups become close to linear. On 64 processors, sorting one million integers takes only 457 milliseconds with a speedup of 56. The speedup curve of sorting 800 K integers on Hector is shown in Figure 4.11 (b). The complexity analysis in Table 4.7 is used for predicting the speedup and execution time. The value of t_r depends on how many processors and which processors are used. The predicted speedups shown match the measured ones within 10% accuracy in Figures 4.11. Since the experimental results on Hector are similar to those on the KSR1, we discuss in detail only the experimental results on the KSR1.

Overhead analysis. Figure 4.12 gives the ratio of the overhead in each phase to the total execution time for one million integers where the overpartitioning ratio is $k = \log_2(P)$ and the oversampling ratio is $s = 3$. Observe that:

- Load imbalance is about 3% and decreases slightly with the number of processors. This indicates that the overpartitioning ratio of $\log_2(P)$ leads to good load balancing.
- Times for selecting pivots and building the task queue grow with the number of processors because their computation times depend on P.
- The total number of remote memory accesses is $\frac{n}{W}$ in the worst case and $\frac{n}{W}\frac{P-1}{P}$ on average. Thus, as the number of processors increases, the fraction of the time involving remote memory accesses increases. When $P > 32$, some portion of remote memory accesses are across ring:1 with a latency of 530 cycles

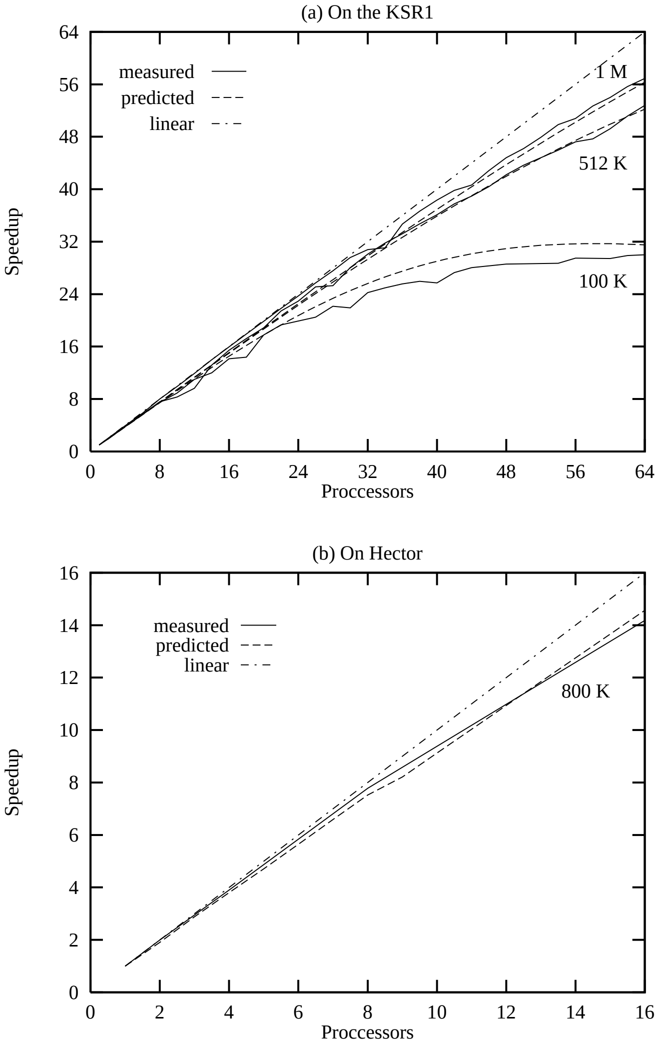


Figure 4.11: Speedups of PQOP

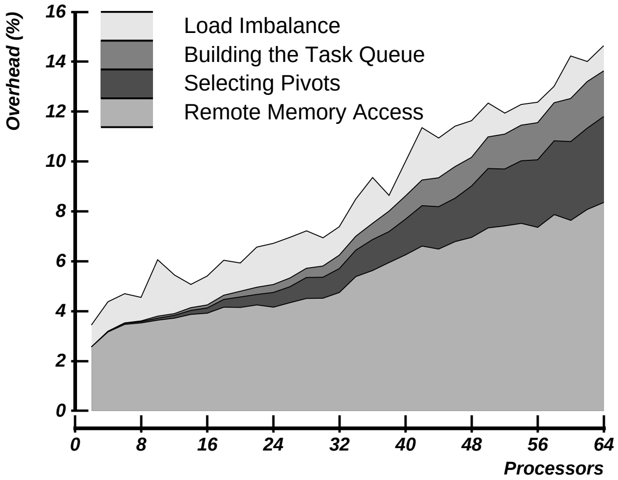


Figure 4.12: Overhead Analysis of PQOP for the KSR1

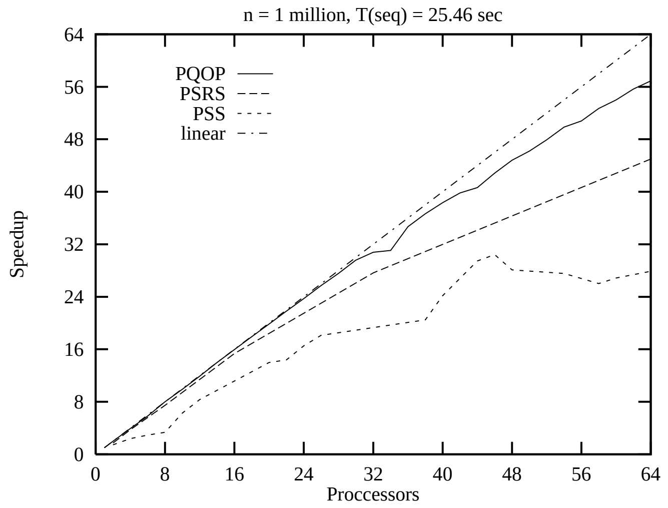


Figure 4.13: Comparison of PSS, PSRS, and PQOP on the KSR1

Algorithm Reference	PQOP	PQOP	PSRS [LLS+93]	PSRS [LLS+93]	Kalé [KK93]	Kalé [KK93]	Tridgell [TB93]	Tridgell [TB93]
Machine	KSR1	KSR1	TC-2000	iPSC/860	iPSC/860	nCUBE/2	CM-5	AP-1000
Processors	64	32	64	64	64	64	32	32
Clock-rate	20	20	20	50	50	20	32	25
Time (sec)	3.78	7.49	4.29	2.46	3.87	12.30	5.48	7.62
Time × Clock-rate	75.6	149.80	85.80	123.00	193.50	246.00	175.36	190.50

Table 4.10: Execution Times for Sorting 8 Million Integers

on the KSR1 (shown in Table 1.1). The number of accesses across ring:1 also depends on the number of processors: $\frac{n}{W}\frac{P-32}{P}$. Thus the time for remote memory accesses increases more quickly when $P > 32$.

Comparison. Two other parallel sorting algorithms Parallel Sample Sort (PSS) and Parallel Sorting by Regular Sampling (PSRS) were also implemented on the KSR1 in order to compare their performance with PQOP. It has been shown that PSS performs well on the SIMD machine CM-2 [BLM+91], and PSRS achieves good performance on a variety of MIMD machines [LLS+93]. Figure 4.13 shows the speedup curves for the three algorithms for sorting one million integers. Although PSS has low communication overhead, it does not balance load well even with an oversampling ratio as large as 32. This results in relatively poor speedups. PSRS minimizes the communication cost and partitions the list into sublists evenly. The speedup of PSRS is much higher than that of PSS, but degrades with the number of processors because:

- The pivoting cost increases with the number of processors.
- Merging used in the last phase of PSRS is not as efficient as binary partitioning based on the data we collected from the KSR1 hardware monitor. With more processors, the time required for merging increases as a percentage of the total execution time and thus accounts for a major portion of the total execution time.

PQOP outperforms both PSS and PSRS on the KSR1. Table 4.10 gives the execution times and speedups of sorting 8 million integers by various algorithms across several machines. Considering the CPU clock rates, the performance of PQOP on the KSR1 is competitive.

Performance of PROP

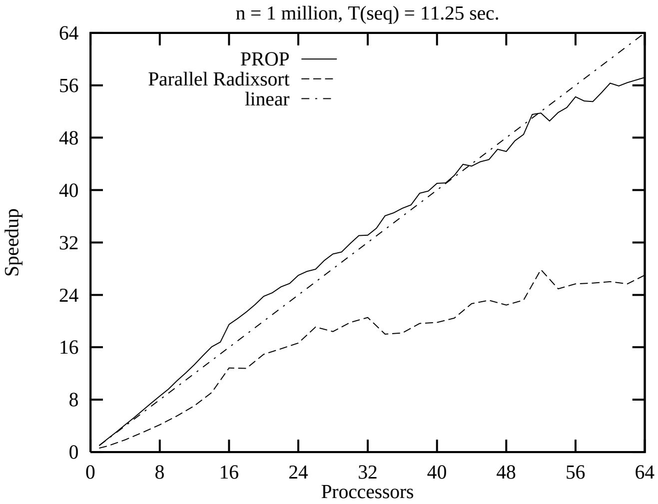


Figure 4.14: Comparison of Parallel Radix Sort and PROP on the KSR1

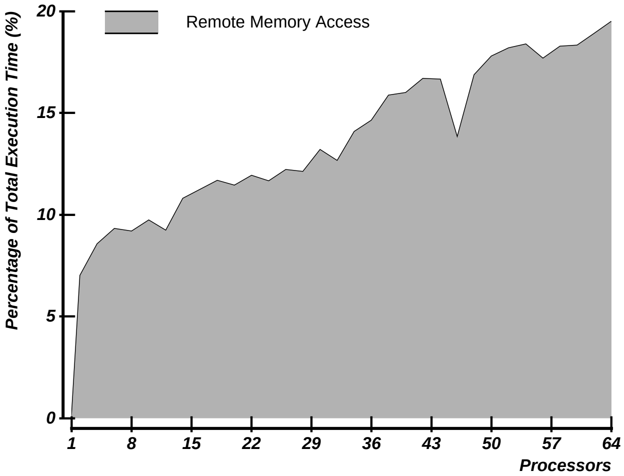


Figure 4.15: Overhead of Remote Memory Accesses in PROP on the KSR1

Machine	KSR1	CM-5
Clock-rate	20	32
Algorithm	PROP	Parallel Radix Sort [TS92]
Time (sec)	10.02	16.7
Time × Clock-rate	200.40	534.4

Table 4.11: Execution Times for Sorting 64 Million Integers on 64 Processors on the KSR1 and the CM5

Speedup. Figure 4.14 shows the speedup curves for Parallel Radix Sort and PROP on the KSR1 in sorting one million randomly generated integers. PROP on one processor takes only 11.25 seconds, much faster than the sequential execution of radix sort (19.22 seconds). This difference results from the higher cache hit ratio in PROP due to overpartitioning. Thus, the speedups in Figure 4.14 are relative to the execution time of PROP on one processor.

PROP achieves nearly linear speedups because it balances the load well by overpartitioning and minimizes the remote memory accesses. The super-unitary linear speedups for 8 through 40 processors result from the higher cache hit ratios. Parallel Radix Sort is a multi-step algorithm. It permutes the keys based on the value of the R bits in each step and moves from the least significant to the most significant with each pass. Therefore, the number of remote memory accesses in each step is close to the total number of remote memory accesses in PROP. In our implementation, R was set to 11, and thus the number of steps was 3. The speedup of Parallel Radix Sort is sub-unitary linear because of the high remote memory access overhead and the low cache hit ratio.

Overhead Analysis. Since PROP does not involve the use of pivots for forming sublists, the main overhead is in the cost of remote memory access. Figure 4.15 shows that the overhead of remote memory accesses increases with the number of processors up to about 20% on 64 processors. However, the speedup is not reduced substantially, because of the increase in the cache hit ratio.

Table 4.11 shows that PROP on the KSR1 outperforms Parallel Radix Sorting on the CM-5 with 64 processors for sorting 64 million uniformly distributed 32-bit integers. The difference is more striking when the clock rates are taken into account.

4.4 Summary

Load imbalance in parallel programs arises from the variance in loop iteration execution times and the variance in partition sizes. In this chapter, we have developed techniques for handling these two categories of load imbalance.

Locality-based Dynamic Loop Scheduling is designed for parallel loops with variance in iteration execution times.

- *Data Locality.* LDS utilizes data distribution information to maximize data locality.
- *Load Balancing.* LDS employs a global dynamic loop scheduling scheme to guarantee good load balance.
- *Low Overhead.* The global dynamic scheduling scheme in LDS minimizes lock operations and avoids an extensive search for work by idle processors.

We have experimentally compared LDS with other loop scheduling schemes in different types of applications. LDS performs at least as well as the best of any competitive scheme in all cases investigated.

Overpartitioning is proposed for problems with a variance in sub-problem size. In parallel sorting, load imbalance and data locality are two critical factors in determining performance. Parallel Sorting by OverPartitioning (PSOP) uses the overpartitioning technique to deal with variance in the partition sizes. We have shown the following properties of PSOP:

- *Data Locality.* PSOP partitions the sublists in a single-step and thus assures that each element moves between processor memories at most once.

- *Load Balancing.* Overpartitioning is a very effective technique and leads to good load balancing with high probability. Overpartitioning leads to much better load balancing than oversampling. Loads on the processors differ by less than 5% in PSOP, and by about 30% in PSS.
- *Nearly Linear Speedups.* PSOP achieves nearly linear speedups and the performance obtained is very close to that predicted by the analytical model. PSOP is shown to outperform other parallel sorts (PSRS, PSS, and Parallel Radix Sort).

Chapter 5

Cache Affinity-based Software Cache Coherence

In the previous chapters, we have focused on reducing the number of unnecessary remote memory accesses from mismatching and false sharing. In Chapters 2 and 3, we have built a framework of data transformations for improving both memory and cache locality. For applications with load imbalance, in Chapter 4, we have proposed Locality-based Dynamic Scheduling (LDS) and Overpartitioning for achieving both load balancing and data locality. In this chapter, we address the issue of coherence cache misses.

Cache coherence is necessary for multiprocessors. Directory-based hardware cache coherence schemes [Ste90] have been used in many multiprocessor systems, such as the KSR1 [KSR91], Dash [LLG+90], and Alewife [ACJ+91]. They provide the programmer with a coherent memory model, but require substantial hardware support. Moreover, several studies have shown that compiler-directed software cache coherence can be as effective as directory-based hardware cache coherence [CV91, CV92, MB90, OA89, AAHV91]. On Cedar [KTV+91], the BBN TC2000 [BBN90], Hector [VSWL91], and the Cray T3D, cache coherence is provided only through software.

Cache affinity involves the reuse of cache contents across parallel loops. The KSR1 Parallel Fortran [KSR93] provides a special directive, *Affinity Region*, to annotate cache affinity. Rothber, Singh, and Gupta [RSG93] show that in most computationally-intensive applications, the working sets are much smaller than the data sets, and they grow slowly with the problem size. Therefore, with increasing cache sizes and numbers of processors, cache affinity becomes more important to performance.

Most current software coherence schemes (such as FSI [CV88a, CV88b] and SCTV [DMCK92]) do not exploit cache affinity. Although some compiler-directed schemes (such as Version Control [CV90], Time Stamp [MB89, MB90], and Life Span [Che92]) attempt to retain cache affinity across parallel loops, they require additional bits to be added into each cache line. Thus they cannot be implemented on current scalable shared-memory multiprocessors (such as the BBN TC2000, Hector, and the Cray T3D).

In this chapter, we first discuss the effect of cache affinity, then propose a Cache Affinity-based Software cache coherence scheme (CAS) and present algorithms for implementing CAS. We evaluate CAS on the Hector multiprocessor, comparing it with several other schemes.

5.1 The Effect of Cache Affinity

We use an example shown in Figure 5.1 to illustrate cache affinity across parallel loops. This example contains two parallel loops enclosed in a sequential loop. Arrays are stored in row-major fashion. The elements of arrays A and B are heavily reused between the two parallel loops. So most cache lines fetched in loop j_1 may still be valid in loop j_2. To exploit such cache affinity (temporal locality), both parallel loops j_1 and j_2 should be scheduled in a block fashion so that the corresponding blocks of these loops can be executed by the same processor. In this way:

```
DO I = 0, Iters
  DOALL j1 = 1, 1000            /* loop j1 */
    DO k = 1, 1000
      A(j1, k) = f(B(j1, k), B(j1 - 1, k), B(j1 + 1, k),
                   B(j1, k - 1), B(j1, k + 1))
    END DO
  END ALL

  DOALL j2 = 1, 1000            /* loop j2 */
    DO k = 1, 1000
      B(j2, k) = A(j2, k)
    END DO
  END ALL
END DO
```

Figure 5.1: An Example of Cache Affinity Across Parallel Loops

	Hardware Support	A	B
Optimal	Full Hardware Directory	1.00	0.99
SCTV	No	0.50	0.80
FSI	1 bit/cache line	0.50	0.80
Life Span	2 bits/cache line	1.00	0.99
Version Control	several bits/cache line	1.00	0.99
Time Stamp	several bits/cache line	1.00	0.99
CAS	No	1.00	0.99

Table 5.1: Estimated Cache Hit Ratios

- Each element of array A is read and written by the same processor, so array A does not cause any cache coherence problem.
- Each element of array B, except those in boundary rows, is also read and written by the same processor. The boundary rows on processor p are shared by processors $p-1$ and $p+1$ in loop j_1. To maintain cache coherence for array B, each processor needs to invalidate two neighboring rows before parallel loop j_1 and to update its two boundary rows after parallel loop j_2.

Assume that the total size of all caches is large enough to accommodate the entire arrays A and B, the cache line size is two words, and each processor executes 100 iterations of each parallel loop. Then under the optimal scheme discussed above, the cache hit ratios for arrays A and B are one and 0.99 respectively. Table 5.1 indicates how various proposed schemes do relative to the optimal.

The SCTV scheme [DMCK92] does not exploit cache affinity across parallel loops. It conservatively assumes that

- every variable (except read-only variables) may be modified by other processors, so upon entering each parallel loop, the cache lines corresponding to the variable are invalidated before the variable is used; and
- every modified variable in the current parallel loop may be used by other processors in the next parallel loop, so the corresponding updated cache lines are written into main memory.

Under SCTV, in loop j_1 of this example, each processor invalidates the rows of array B to be read before loop j_1 and writes back the modified rows of array A after loop j_1. In loop j_2, similarly, each processor

```
DO I = 1, N
  DOALL J1 = ...
    ...
  END ALL
  DOALL J2 = ...
    ...
  END ALL
  ...  ...
  DOALL Jn = ...
    ...
  END ALL
END DO
```

Figure 5.2: Loop Structure

invalidates the rows of array A to be read and writes back the modified rows of array B. Thus, no cached data can be reused between these two parallel loops. In this example, each cache line for array B is accessed approximately eight times in loop j_1 and twice in loop j_2. Thus, the cache hit ratio is 8/10 (see Table 5.1).

The FSI scheme [CV88a] uses an extra bit in each cache line to indicate valid cache lines within parallel loops, but it cannot exploit cache affinity beyond a single parallel loop. So FSI obtains the same cache hit ratios as those in SCTV. To exploit cache affinity across parallel loops, the Life Span strategy adds two bits to each cache line to identify the cache lines accessed in the last parallel loop that can be used in the current loop. Since the cache affinity in the example is between the two parallel loops, the Life Span strategy [Che92] can also capture such cache affinity and achieve the same cache hit ratios as those in the optimal scheme. The Version Control and the Time Stamp strategies [CV90, MB89] use several bits per cache line, thus they can retain cache affinity across many parallel loops. The Cache Affinity-based Software cache coherence scheme (CAS) captures cache affinity across parallel loops by software. We will discuss it in the next section.

In summary, by considering cache affinity across parallel loops, the cache hit ratios for arrays A and B are increased by 100% and 24% respectively. In the next section, we propose a new scheme that achieves these gains without requiring any hardware support.

5.2 Cache Affinity-based Scheme

In this section, we present the Cache Affinity-based Software cache coherence scheme (CAS). We first describe assumptions and introduce notation, then define the CAS scheme.

5.2.1 Assumptions and Notation

We consider systems in which each processor has a private cache without hardware cache coherence support, but the memory management unit (MMU) provides three cache operations: *invalidate*, *update*, *flush*, which operate on the private cache at the location indicated by a virtual address and with a size of a cache line, a page, or the entire cache. Typically, the costs of these operations are not linear with the amount of data affected. Thus, the overhead can be reduced by aggregating operations on small units into fewer operations on larger units.

We assume that parallelism is expressed strictly via `DOALL` parallel loops, in which there are no data dependences between their iterations [Wol89b]. The general form of loop nest treated in this chapter is shown in Figure 5.2. The outermost sequential loop contains several parallel loops, where each parallel loop may contain a sequence of sequential loop nests. The outermost loop bounds are constants, inner loop bounds are linear functions of outer loop index variables, and all strides are constants. We further assume

j_1	$Def(B, j_1, 2, p_1)$	empty
	$Def(B, j_1, 2, \overline{p_1})$	empty
	$Use(B, j_1, 2, p_1)$	rows 100 to 201
	$Use(B, j_1, 2, \overline{p_1})$	rows 0 to 101, 200 to 1000
	$Ref^{down}(B, j_1, 2, p_1)$	rows 101 to 200
	$Def^{up}(B, j_1, 2, p_1)$	rows 101 to 200
j_2	$Def(B, j_2, 2, p_1)$	rows 101 to 200
	$Def(B, j_2, 2, \overline{p_1})$	rows 1 to 100, 201 to 1000
	$Use(B, j_2, 2, p_1)$	empty
	$Use(B, j_2, 2, \overline{p_1})$	empty
	$Ref^{down}(B, j_2, 2, p_1)$	rows 100 to 201
	$Def^{up}(B, j_2, 2, p_1)$	rows 101 to 200

Table 5.2: Data Sets for the Example in Figure 5.1

that array indices are linear functions of loop index variables. Read-only arrays and scalar variables are not discussed because read-only arrays do not introduce cache coherence problems. Since multiple arrays can be treated one by one, for simplicity, we assume that there is only one array A in the loop nest and it appears in every parallel loop. If array A is not used in some parallel loops, these loops can be ignored in applying the cache coherence algorithm.

Let $Inv(A, J_k, I, p)$ and $Upd(A, J_k, I, p)$ respectively represent the minimum portions of the array which have to be invalidated before and updated after parallel loop J_k of the I^{th} iteration of the sequential outermost loop on processor p. To compute $Inv(A, J_k, I, p)$ and $Upd(A, J_k, I, p)$, we introduce the following notation similar to that used in data flow analysis [ASU86]:

- $Use(A, J_k, I, p)$ and $Def(A, J_k, I, p)$ are the portions of array A read/written in parallel loop J_k within the sequential loop I^{th} iteration by processor p respectively, and $Ref(A, J_k, I, p)$ is defined as $Use(A, J_k, I, p) \cup Def(A, J_k, I, p)$;
- $Use(A, J_k, I, \overline{p})$ and $Def(A, J_k, I, \overline{p})$, are the portions of array A read/written in parallel loop J_k within the sequential loop I^{th} iteration by processors other than p respectively, and $Ref(A, J_k, I, \overline{p})$ is $Use(A, J_k, I, \overline{p}) \cup Def(A, J_k, I, \overline{p})$;
- $Ref^{down}(A, J_k, I, p)$ is the portion of array A referenced (*i.e.*, read or written) before parallel loop J_k in the sequential loop I^{th} iteration by processor p, but which has not been written by other processors after the references until the parallel loop;
- $Def^{up}(A, J_k, I, p)$ is the portion of array A written after parallel loop J_k in the sequential loop I^{th} iteration by processor p, but which has not been referenced by other processors from the parallel loop until the writes; and
- $Use^{up}(A, J_k, I, \overline{p})$ is the portion of array A read after parallel loop J_k in the sequential loop I^{th} iteration by processors other than p, but which has not been written by processor p from the parallel loop until the references.

To illustrate these definitions, we list in Table 5.2 the data sets for the example in Figure 5.1, where processor 1 executes iterations 101 to 200 in both of the parallel loops j_1 and j_2.

5.2.2 The Precise Algorithms

To maintain cache coherence for array A in parallel loop J_k in the I^{th} iteration of the outermost sequential loop, it is sufficient that the portion of array A to be read (*i.e.*, $Use(A, J_k, I, p)$) is invalidated before the parallel loop and the portion of array A written (*i.e.*, $Def(A, J_k, I, p)$) is copied back to main memory after the parallel loop. Under this conservative scheme, however, data cannot be reused in the cache across parallel loops.

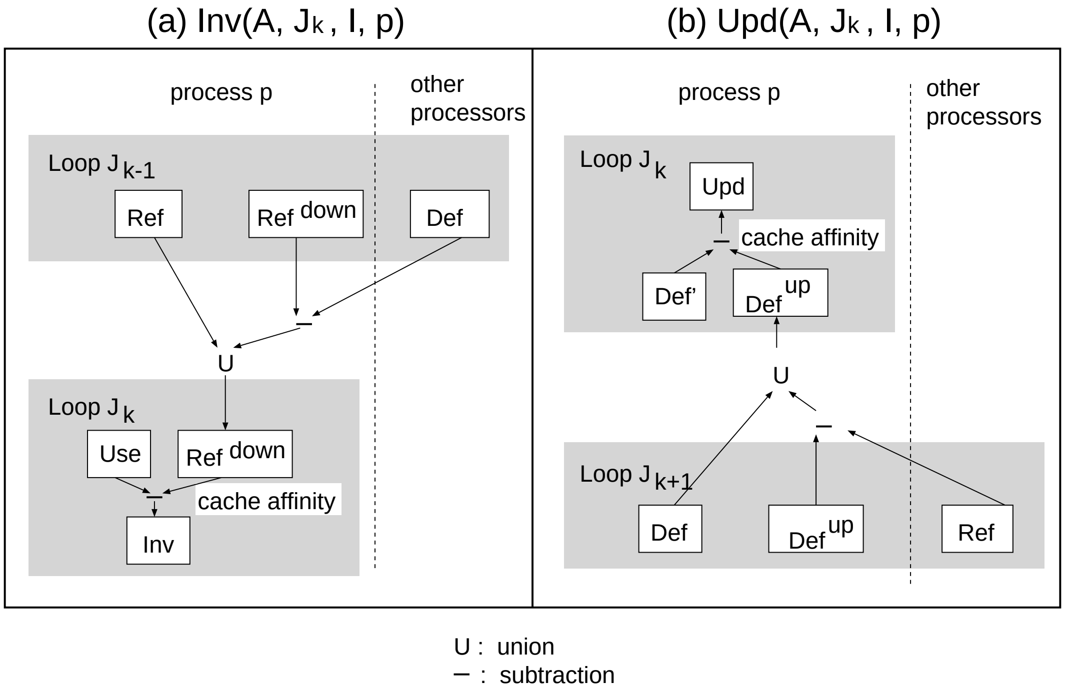


Figure 5.3: Calculations of Invalidation and Update Sets under CAS

The cache affinity for parallel loop J_k can be precisely defined in terms of $Ref^{down}(A, J_k, I, p)$ and $Def^{up}(A, J_k, I, p)$.

- $Ref^{down}(A, J_k, I, p)$ represents the valid data set on processor p before parallel loop J_k, so the cache lines for the data set can be reused in parallel loop J_k by processor p.
- $Def^{up}(A, J_k, I, p)$ represents the data set that will be rewritten by processor p before being referenced by other processors, so the dirty cache lines for the data set do not need to be written back to the main memory.

We propose a Cache Affinity-based Software cache coherence scheme (CAS) to exploit cache affinity by avoiding unnecessary invalidations and updates as follows (see Figure 5.3):

$$Inv(A, J_k, I, p) = Use(A, J_k, I, p) - Ref^{down}(A, J_k, I, p) \quad (5.1)$$

$$\begin{aligned} Upd(A, J_k, I, p) &= Def(A, J_k, I, p) \cap \\ &\quad Use^{up}(A, J_k, I, \overline{p}) - Def^{up}(A, J_k, I, p) \end{aligned} \quad (5.2)$$

where "–" is the subtraction operation on sets. In Eq. (5.1), $Ref^{down}(A, J_k, I, p)$ represents the useful cache context for loop J_k that need not be invalidated. In Eq. (5.2), the term $Def^{up}(A, J_k, I, p)$ prevents redundant updates by the same processor, and the term $Use^{up}(A, J_k, I, \overline{p})$ prevents unnecessary updates where other processors do not use the updated values. Because of the complexity of $Use^{up}(A, J_k, I, \overline{p})$, we replace it by the total data set read by other processors after parallel loop J_k, $\sum_{i=I}^{N} \sum_{j=k+1}^{n} Use(A, J_j, i, \overline{p})$. Then Eq. (5.2) becomes:

$$\begin{aligned} Upd(A, J_k, I, p) &\subseteq Def(A, J_k, I, p) \cap \\ &\quad \sum_{i=I}^{N} \sum_{j=k+1}^{n} Use(A, J_j, i, \overline{p}) - Def^{up}(A, J_k, i, p) \end{aligned} \quad (5.3)$$

<table>
<tr><th></th><th>$Ref^{down}(A, J_k, I, p)$</th><th>$Def^{up}(A, J_k, I, p)$</th></tr>
<tr><td>$k = 1$</td><td>$Ref(A, J_n, I-1, p) \cup$ $(Ref^{down}(A, J_n, I-1, p) -$ $Def(A, J_n, I-1, \overline{p}))$</td><td rowspan="2">$Def(A, J_{k+1}, I, p) \cup$ $(Def^{up}(A, J_{k+1}, I, p) -$ $Ref(A, J_{k+1}, I, \overline{p}))$</td></tr>
<tr><td>$1 < k < n$</td><td rowspan="2">$Ref(A, J_{k-1}, I, p) \cup$ $(Ref^{down}(A, J_{k-1}, I, p) -$ $Def(A, J_{k-1}, I, \overline{p}))$</td></tr>
<tr><td>$k = n$</td><td>$Def(A, J_1, I+1, p) \cup$ $(Def^{up}(A, J_1, I+1, p) -$ $Ref(A, J_1, I+1, \overline{p}))$</td></tr>
</table>

Table 5.3: Computing Ref^{down} and Def^{up}

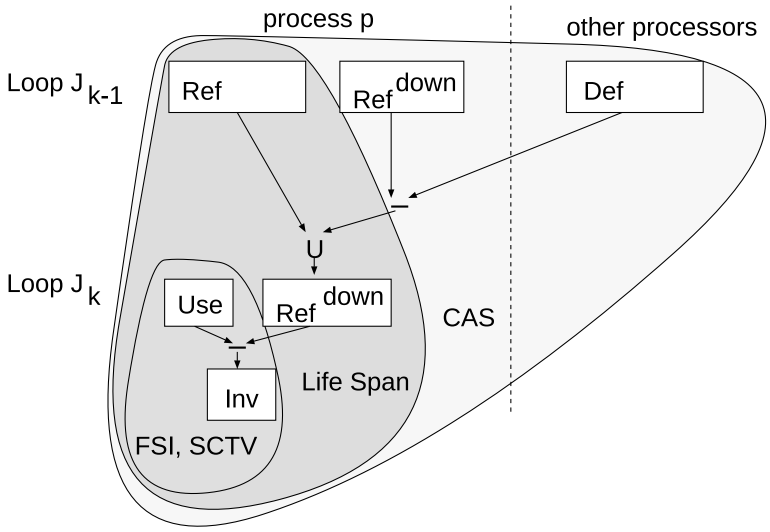


Figure 5.4: Cache Affinity in Software Cache Coherence Schemes

The remaining issue is to compute the cache affinity (Ref^{down} and Def^{up}) for each parallel loop. Figure 5.3 shows that Ref^{down} can be computed forwards and Def^{up} backwards from the Def and Use sets, respectively ($Def' = Def(A, J_k, I, p) \cap \sum_{i=I}^{N} \sum_{j=k+1}^{n} Use(A, J_j, i, \overline{p})$). The detailed calculations are specified in Table 5.3. For parallel loop J_k ($k > 1$), $Ref^{down}(A, J_k, I, p)$ may include

- $Ref(A, J_{k-1}, I, p)$: the data referenced by processor p in parallel loop J_{k-1}, and
- $Ref^{down}(A, J_{k-1}, I, p)$: the valid data set before parallel loop J_{k-1}.

However, the valid data set on processor p should not include the data set written by other processors in parallel loop J_{k-1}, $Def(A, J_{k-1}, I, \overline{p})$. Since

$$Def(A, J_{k-1}, I, \overline{p}) \cap Ref(A, J_{k-1}, I, p) = \emptyset,$$

we do not need to subtract the Def set from the Ref set. If there exist control dependences in the loop nest, the data flow analysis for array accesses proposed by Granston and Veidenbaum [GV91] can be incorporated into our algorithms.

Figure 5.4 shows the relationships with respect to cache affinity for invalidation among several software cache coherence schemes. The SCTV and FSI schemes consider only single parallel loops without exploiting cache affinity across parallel loops. The Life Span strategy uses a *change* bit and a *stable* bit so that data

<table>
<tr><th></th><th>$Ref_{min}^{down}(A, J_k, I, p)$</th><th>$Def_{min}^{up}(A, J_k, I, p)$</th></tr>
<tr><td>$k = 1$</td><td>$Ref_{min}(A, J_n, I-1, p) \cup_{min}$
$(Ref_{min}^{down}(A, J_n, I-1, p) -_{min}$
$Def_{max}(A, J_n, I-1, \overline{p}))$</td><td rowspan="2">$Def_{min}(A, J_{k+1}, I, p) \cup_{min}$
$(Def_{min}^{up}(A, J_{k+1}, I, p) -_{min}$
$Ref_{max}(A, J_{k+1}, I, \overline{p}))$</td></tr>
<tr><td>$1 < k < n$</td><td rowspan="2">$Ref_{min}(A, J_{k-1}, I, p) \cup_{min}$
$(Ref_{min}^{down}(A, J_{k-1}, I, p) -_{min}$
$Def_{max}(A, J_{k-1}, I, \overline{p}))$</td></tr>
<tr><td>$k = n$</td><td>$Def_{min}(A, J_1, I+1, p) \cup_{min}$
$(Def_{min}^{up}(A, J_1, I+1, p) -_{min}$
$Ref_{max}(A, J_1, I+1, \overline{p}))$</td></tr>
</table>

Table 5.4: Approximate Methods for Ref_{min}^{down} and Def_{min}^{up}

accessed in the last parallel loop need not be invalidated. So its cache affinity is the data set accessed by the same processor in the last parallel loop. The Version Control and the Time Stamp strategies retain cache affinity across many parallel loops. So their cache affinity can be the same as in the CAS scheme. However, they require hardware support.

5.2.3 The Approximate Algorithms

In Section 5.2.2, we assume that Def and Use sets can be accurately obtained and that operations on these sets are closed. However, if we use regular sections [Bal90, HK91] to represent Use and Def data sets, neither of these two assumptions holds. Therefore, we relax the assumptions and introduce approximate algorithms.

Let us use subscripts max and min to represent supersets and subsets. For example, Def_{max} is a superset of Def such that $Def_{max} \supseteq Def$. We also introduce the following approximate set operations:

- "$\cup_{max}$", "$\cup_{min}$" give a superset and a subset of the union of two sets respectively.
- "$-_{max}$", "$-_{min}$" give a superset and a subset of the difference between two sets respectively.
- "$\cap_{max}$" gives a superset of the intersection of two sets.

We use Inv_{max} and Upd_{max} to represent the data sets which will be invalidated and updated respectively in our approximate scheme. They can be computed by the following set of equations:

$$Inv_{max}(A, J_k, I, p) = Use_{max}(A, J_k, I, p) -_{max} Ref_{min}^{down}(A, J_k, I, p) \tag{5.4}$$

$$Upd_{max}(A, J_k, I, p) = Def_{max}(A, J_k, I, p) \cap_{max} \sum_{i=I}^{N} \sum_{j=k+1}^{n} Use(A, J_j, i, \overline{p}) -_{max} Def_{min}^{up}(A, J_k, I, p) \tag{5.5}$$

Using the approximate operations and sets, we can derive from Table 5.3 approximate methods for computing Ref_{min}^{down} and Def_{min}^{up} as shown in Table 5.4.

5.3 Parametric Section Analysis

In Section 5.3, we have developed the CAS cache coherence scheme. In this section, we discuss the details of this scheme. We first extend bounded regular sections to parametric regular sections, then use parametric regular sections to summarize array references using Def^{min}, Def^{max}, Use^{min}, and Use^{max} for each parallel loop. Finally, we define approximate set operations on parametric regular sections used by the approximate algorithm given in Section 5.3.3.

5.3.1 Parametric Sections

Bounded regular sections [HK91] use the triplet notation $[l : u : s]$ to represent a rectangular array region, where l and u are often called the lower and upper bounds respectively, and s is the stride. The data set of a reference in a loop can be summarized by a bounded regular section. For example, in the loop:

```
DO J = L, U, S
    A(2*J + 1) = B(J, J)
END DO
```

the data set accessed by $A(2*J+1)$ is represented as $A[(2*L+1) : (2*U+1) : (2*S)]$.

Bounded regular section analysis has been used in inter-procedural analysis [HK91] and in data distribution analysis [HKT91]. To use it in the CAS scheme, we need to make some extensions for the following reasons:

- The algorithms for bounded section analysis give only supersets of sections for inter-procedural data dependence analysis [HK91]. However, our approximate cache coherence algorithms in the CAS scheme require both supersets and subsets of sections.
- The previous algorithms assume that the upper and lower bounds of sections are constants [HK91]. Since a parallel loop is scheduled among processors, parallel loop index variables should remain in sections for computing Inv and Upd sets so that the cache affinity on each processor can be determined.

We propose *parametric sections* that allow the index variables of the sequential loop and the parallel loop to appear in their upper and lower bounds. Parallel sections are in the form of $[L : U : s]$, where $L = a_l I + b_l J + c_l$ is the lower bound, $U = a_u I + b_u J + c_u$ is the upper bound, I and J are the index variables of the sequential loop and the current parallel loop respectively, and $a_l, b_l, c_l, a_u, b_u, c_u$, and s are constants. If both b_l and b_u are zero, the section is independent of the parallel loop, and thus is called a *sequential* (parametric) section. A parametric section without index variables I and J (*i.e.* where $a_l = b_l = a_u = b_u = 0$) degenerates to a bounded regular section.

Two parametric sections in the same parallel loop are said to be in the same *parallel class* if they have equal a_l's and equal a_u's. They are called in the same *class* if they also have equal b_l's and equal b_u's. Sections within the same (parallel) class have some potential cache affinity. For example, consider $S_1 = [2I + 3J : 3I + 4J : 1]$, $S_2 = [2I + 3J : 3I + 4J + 3 : 3]$, and $S_3 = [2I + 3J : I + 3J : 1]$. Then S_1 and S_2 are in the same class, but S_3 is in the same parallel class with S_1 and S_2, not in the same class.

We use parametric sections in our cache coherence algorithms. The parametric section analysis consists of two major steps. The first step represents the portion of an array accessed by each reference using a parametric section, and merges sections that are in the same class. The second step identifies cache affinity between parallel loops and computes Inv and Upd.

5.3.2 Summarizing References Using Parametric Sections

For each parallel loop, read and write references to each array are classified into four sets:

$$Use_{max}, Use_{min}, Def_{max}, Def_{min}$$

These sets are represented by a list of parametric sections. For a reference to a one-dimensional array, we compute two array sections, a superset and a subset of the array region touched by the reference (S_{max} and S_{min}), based on loop bounds and loop strides. If the reference is a read, S_{max} and S_{min} are added to Use_{max} and Use_{min} respectively. Otherwise, they are added to Def_{max} and Def_{min}. For a reference to a multi-dimensional array, we compute array sections for each dimension independently. If the index expressions of the reference are coupled, the array section for the subset, S_{min}, is assumed to be empty.

After references are summarized as parametric sections, two sections (S_1 and S_2) in the same class are merged if they satisfy one of the following five conditions:

1. $S_1 \subseteq S_2$
2. $S_2 \subseteq S_1$

```
real A[N][N]
DO 10 i = 1, N
   DOALL j = i+1, N
      A(j, i) = A(j, i)/A(i, i)
      DO k = i+1, N
         A(j, k) = A(j, k) - A(j, i)*A(i, k)
      END DO
   END ALL
END DO
```

Reference	Section	Merged Section
A(i, i) A(i, k)	A[i:i:1][i:i:1] A[i:i:1][i+1:N:1]	A[i:i:1][i:N:1]
A(j, i) A(j, k)	A[j:j:1][i:i:1] A[j:j:1][i+1:N:1]	A[j:j:1][i:N:1]

Table 5.5: Summarizing References in LU

3. $S_1||S_2$: they can be concatenated if
   - they have the same stride,
   - the difference between their lower bounds is a multiple of the stride, and
   - the upper bound of S_1 is not less than the lower bound of S_2.
4. $S_2||S_1$: similar to 3.
5. S_1 and S_2 have the same (or close) upper and lower bounds, but with different strides. Since strides are usually small constants, we use bit vectors to represent strides. So the merging in this case only requires setting some bits in the bit vector.

Table 5.5 summarizes the sections for read references in the parallel LU decomposition. The supersets and subsets are the same. Among these sections, $A[i:i:1][i:i:1]$ and $A[i:i:1][i+1:N:1]$ are in one class, and $A[j:j:1][i:i:1]$ and $A[j:j:1][i+1:N:1]$ are in the other, and they are merged into $A[i:i:1][i:N:1]$ and $A[j:j:1][i:N:1]$, respectively.

5.3.3 Approximate Operations for the CAS Scheme

In the CAS scheme, the Inv and Upd sets can be computed from the Use and Def data sets using equations in Section 5.2.3. These equations require the following approximate operations on parametric sections: "$\cup_{min}$", "$-_{min}$", and "$-_{max}$".

The "$-_{min}$" and "$\cup_{min}$" operations are used in computing both $Ref^{down}_{min}(A, J_k, I, p)$ and $Def^{up}_{min}(A, J_k, I, p)$. Since they are applied to the parametric sections from different parallel loops, cache affinity between parallel loops should be taken into account.

Consider two parametric sections from parallel loops J_k and J_l:

$$S_1 = [a^1_l I + b^1_l J_k + c^1_l : a^1_u I + b^1_u J_k + c^1_u : s_1]$$

$$S_2 = [a^2_l I + b^2_l J_l + c^2_l : a^2_u I + b^2_u J_l + c^2_u : s_2]$$

The two sections have "good" cache affinity if the following two conditions are satisfied:

- The loop scheduling schemes for the two parallel loops retain affinity. Three scheduling schemes are considered: block, cyclic, and block-cyclic. Block scheduling will be used to illustrate our method. To preserve affinity in block scheduling for parallel loops J_k and J_l, iterations J_k and $J_l + D$ are mapped onto the same processor, where D is a small constant such that $J_k = J_l + D$
- The two sections are in the same parallel class so that they can share most of the data.

In both Ref_{min}^{down} and Def_{min}^{up}, we need to compute parametric sections from different loops: $S_1(p) -_{min} S_2(\overline{p})$, where $S_1(p)$ is the portion of the section S_1 on processor p and $S_2(\overline{p})$ is the portion of the section S_2 on processors other than p.

- If $S_1(p)$ and $S_2(\overline{p})$ do not have "good" cache affinity, we can simply check whether $S_1(p)$ and $S_2(\overline{p})$ are disjoint. The result is set to $S_1(p)$ if they are disjoint, or $\emptyset$ otherwise.
- If they have "good" cache affinity, *i.e.*, $S_1(p)$ and $S_2(\overline{p})$ are in the same parallel class and use block scheduling such that $J_k = J_l + D$, then $S_2(\overline{p})$ may share a small portion of $S_1(p)$. For example, $S_1 = [j : j : 1]$ and $S_2 = [j+1 : j+1 : 1]$, then $S_1(p) -_{min} S_2(\overline{p})$ is close to $S_1(p)$ except that the lower bound will be increased by $D+1$ if $D+1 > 0$, or the upper bound will be reduced by $D+1$ otherwise. So a pair of such offsets, $F = (f_l, f_u)$, are added to the specification of parametric sections in the form of $[L : U : s : F]$.

To compute $S(p) = S_1(p) \cup_{min} S_2(p)$, we extend the merging algorithm in section 3.2 by considering the offsets.

The last operation in computing $Inv_{max}(A, J_k, I, p)$ is "$-_{max}$":

$$Inv_{max}(A, J_k, I, p) = Use_{max}(A, J_k, I, p) -_{max} Ref_{min}^{down}(A, J_k, I, p)$$

We first check the offsets of parallel sections in Ref_{min}^{down}. If they intersect with Use_{max}, the offsets should be in the invalidation set. We then find out the difference ("$-_{max}$") between sections in Use_{max} and Ref_{min}^{down} without considering offsets, which is similar to computing "$-_{min}$".

Computing Upd requires examining whether the data written in the current parallel loop will be read by other processors: $\sum_{i=I+1}^{N} \sum_{j=k+1}^{n} Use(A, J_j, I, \overline{p})$ (see Eq. 5.5). It is not practical to compute the summation over all the sequential iterations. To simplify the task, we classify the sections in $Use(A, J_j, i, \overline{p})$ into two groups based on whether they contain the outermost sequential loop index variable. All the sections without the outermost sequential loop index variable are moved out from the summation. For a section with the sequential index variable, we determine whether the section increases or decreases with the index variable.

The last step is to generate code for invalidations and updates. For the invalidation sections defined by offsets, we merge them to obtain the largest offsets for the current parallel loop, and insert *invalidate* immediately before the parallel loop. For a sequential section in Inv, invalidating the section is necessary on every processor. For a sequential section in Upd, *update* is inserted inside of the parallel loop, but a guard condition is added before the *update* for each processor to check whether the sequential section is modified in the iterations of the parallel loop scheduled on the processor.

Figure 5.5 gives the parallel LU decomposition after applying our CAS method. The read sections in iteration i include

$$A[j : j : 1][i : N : 1], \text{ and } A[i : i : 1][i : N : 1]$$

where the second one is the pivot row. Clearly, the first section is in the reference set of iteration $i - 1$, so only the pivot row may need to be invalidated. For *update*, the write section is $A[j : j : 1][i : N : 1]$ in iteration i, where all rows except the first in the section will be written by the same processor in iteration $i + 1$. This row is thus updated immediately within the parallel loop.

5.4 Cache Line False Sharing

The cache line size on most machines is much larger than a single word. When a cache line is written by several processors within a parallel loop, in the absence of corrective measures, one processor will successfully

```
invalidate(&A(0,0), sizeof(real)*N*N)
DO 10 i = 1, N
   invalidate(&A(i, i), sizeof(real)*(N-i))
   DOALL j = i+1, N
      A(j, i) = A(j, i)/A(i, i)
      DO k = i+1, N
         A(j, k) = A(j, k) - A(j, i)*A(i, k)
      END DO
      IF (j == (i+1))
      THEN update(&A(i+1, i+1), sizeof(real)*(N-i))
      END IF
   END ALL
END DO
update(&A(0, 0), sizeof(real)*N*N)
```

Figure 5.5: LU Decomposition after Applying CAS

write its updated value in the cache line into main memory by an *update* operation, and other writes will be lost. Therefore, cache line false sharing causes incorrect results. Several schemes such as SCTV and FSI avoid false sharing by using a write-through cache policy, which makes writes slower. We reduce false sharing in CAS as follows:

1. All dimension sizes are enlarged to be a multiple of the cache line size.
2. If there is at least one dimension in which the index expressions do not contain the parallel loop index, this dimension is permuted to the last dimension in order to avoid false sharing.
3. If the parallel loop index appears in every dimension, but there is one of the dimensions where the index expressions are the same, then this dimension is permuted to the last dimension so that the parallel loop can be partitioned to ensure writes be cache line aligned on every processor.
4. If alternatives 2 and 3 fail, we identify false shared pages (cache lines) and set these pages to the write-through policy.

5.5 Evaluation of CAS

The CAS, SCTV, and FSI schemes were applied to three applications, LU decomposition, Successive Over Relaxation (SOR), and the Conjugate Gradient program (CG). These applications were then implemented on the Hector multiprocessor. Our experiments were conducted on a 16-processor, 4-station configuration. The cache line size is 16 bytes. Hector provides the *invalidate, update* and *flush* cache operations for software to maintain coherence. The costs of *invalidate* and *update* on Hector are given in Figure 5.6. The invalidation cost is relatively insensitive to data size and location, but the update cost increases with the data size. Cache hit ratio and speedup were used as performance metrics.

The FSI method uses an extra bit per cache line to exploit locality between iterations of a parallel loop and some additional hardware to invalidate in constant time. On Hector, invalidating the entire cache takes only 90 μs, so that approach can be used as the invalidation for FSI. Two extreme cases were investigated: the worst case is the No Cache version in which arrays are set to uncacheable, and the optimal case is the No Coherence version in which data in caches will never be written back to memory except for evictions (so the results of the programs are incorrect). We first present cache hit ratios for the three applications under different cache coherence schemes, then discuss their speedups on Hector.

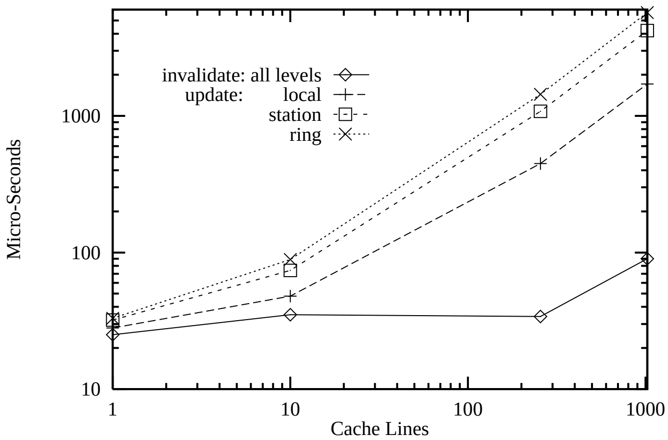


Figure 5.6: Cost of Cache Coherence Primitives (in μs)

5.5.1 Cache Hit Ratios

Cache foot-prints of these applications were collected using Hector's system calls and cache hit ratios were estimated from the collected data and the spatial locality.

LU Decomposition. Figure 5.7 gives the cache hit ratios for LU decomposition with a matrix size of 100×100. At each elimination step, the pivot row is used by all processors to reduce their own rows. So the pivoting row in the i^{th} elimination step is reused $(N - i)/P$ times within parallel loops by every processor. All three methods can utilize such locality in the cache. The cache hit ratios for pivoting rows are shown in Figure 5.7 (a). Other rows are used only once in each elimination step, but reused between elimination steps. Both FSI and SCTV fail to exploit such temporal locality across parallel loops, resulting in a cache hit ratio of .50 from spatial locality. By applying parametric section analysis, we can recognize sharing patterns and fully utilize the cache reuse so that the CAS scheme delivers a higher cache hit ratio. The 12 peaks indicate that there are 12 phases of cache conflict between pivoting rows and others when eight processors are used for LU decomposition of a matrix of size 100×100. When 16 processors are used, the number of peaks is reduced to six (Figure 5.7 (b)).

Figure 5.7 (b) shows the cache hit ratios for non-pivoting rows for from 1 to 16 processors. When fewer processors are used, the caches cannot accommodate the working set (the current pivoting row and the currently used non-pivoting rows). Figure 5.7 (b) shows that, in the first stages of the LU decomposition, the cache hit ratios for non-pivoting rows is close to zero when the working set is much larger than the cache size, and the cache hit ratios are increased when the working set size is decreased.

SOR. Figure 5.8 gives cache hit ratios for square matrix sizes of 128 and 256. In the FSI and SCTV schemes, only spatial locality can be exploited for both sizes, so the cache hit ratio is estimated as 0.5. In the CAS scheme, data can be reused across parallel loops. However, the cache hit ratios depend on whether the cache size is large enough to hold the working set. For a matrix size of 128, the working set is 16K when eight processors are used. So the cache hit ratio is 0.5 for fewer than eight processors. The cache hit ratios decrease when the number of processors is greater than eight, because the remote memory access ratios increase with the number of processors. For the matrix size of 256, the cache hit ratio is also .5 for fewer than 16 processors since the working set is larger than the cache size. On 16 processors, the cache hit ratio for 256 is higher than that for 128, because the ratio of the number of remote memory accesses to the number of local memory accesses decreases with the problem size.

CG. Figure 5.9 gives cache hit ratios for vectors used in CG. It does not include cache hit ratios for the sparse matrix since the matrix is read-only and causes no cache coherence problems. Accesses to these vectors within parallel loops retain spatial locality, so both SCTV and FSI obtain a cache hit ratio of 0.5. The CAS method achieves a higher cache hit ratio by exploiting temporal locality across parallel loops.

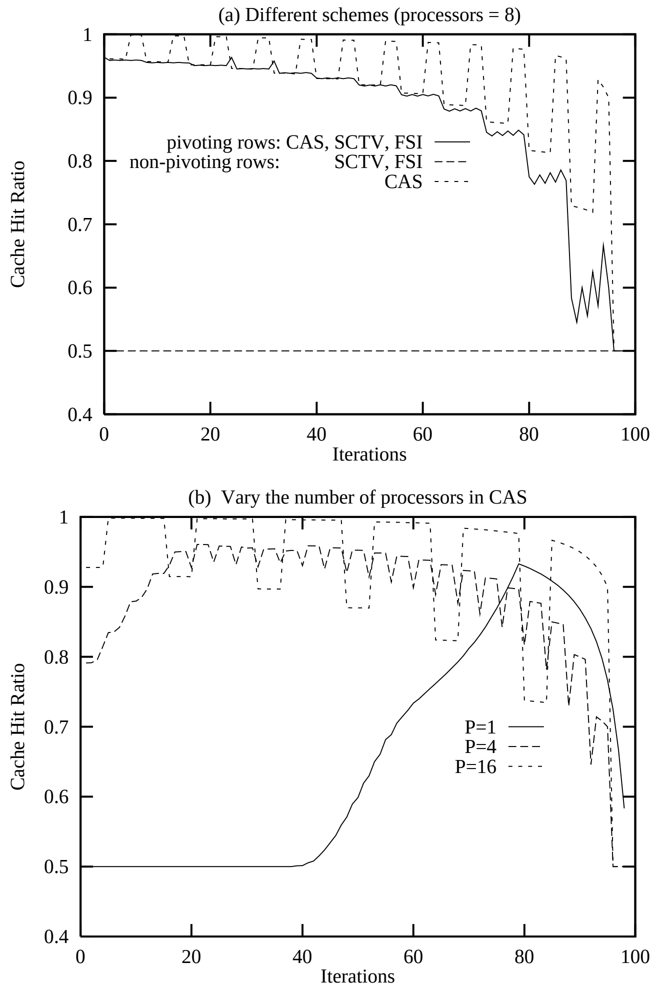


Figure 5.7: Cache Hit Ratios in LU Decomposition with $N = 100$

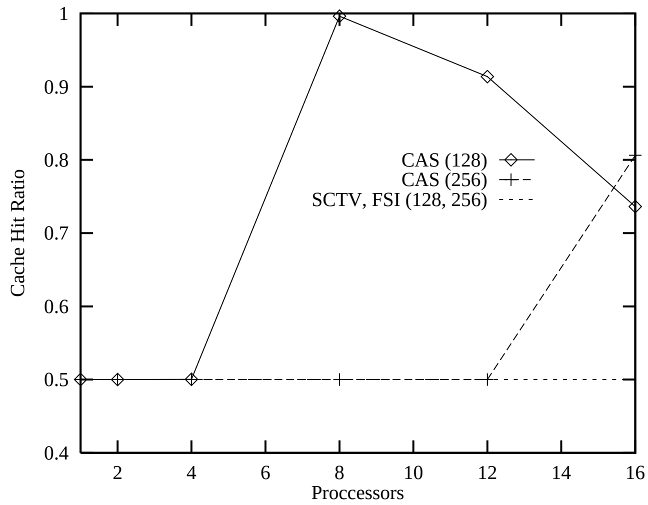


Figure 5.8: Cache Hit Ratios in Successive Over Relaxation with $N = 128$ or 256

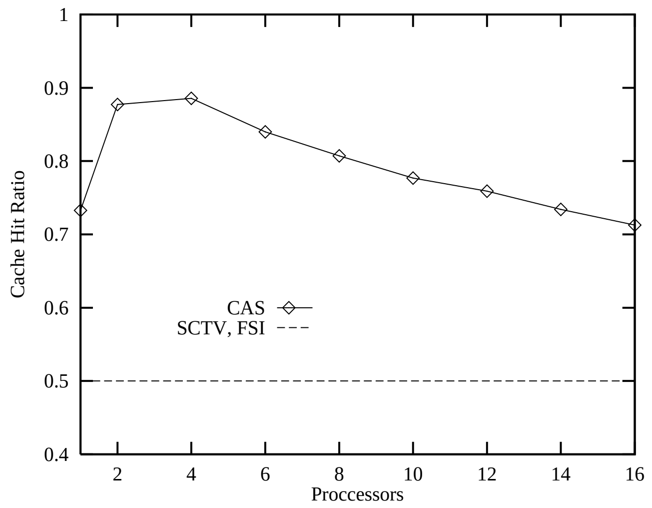


Figure 5.9: Cache Hit Ratios in the Conjugate Gradient with $N = 1400$ and 78148 Non-zeros

5.5.2 Speedups

Figures 5.10, 5.11, and 5.12 show speedups attained by the three applications. The speedups are calculated relative to the execution time of the sequential version.

LU Decomposition. The speedups of LU decomposition are shown in Figures 5.10. The No Cache version obtains limited speedups (less than two) for all the problem sizes. Both FSI and SCTV improve performance significantly over the No Cache version because they exploit spatial locality. Although they have the same cache hit ratio, the speedup in FSI is slightly higher because the overhead of invalidating the entire cache in FSI is less than that of invalidating individual rows in SCTV. The CAS version retains data locality across the elimination steps, thus outperforming FSI and SCTV. When the problem size increases, the performance of CAS becomes close to that of the No Coherence version, which is the upper bound. Once the problem size is sufficiently big so that the cache cannot hold the working set (for example, a matrix size of 300), all these schemes perform similarly, since no temporal locality can be exploited for non-pivot rows across the elimination steps.

SOR. Speedups of SOR for matrix sizes of 128 and 256 are shown in Figure 5.11. For the size of 128, all schemes except No Cache perform equally well when the number of processors is less than eight, because no temporal locality across parallel loops can be found in the cache. When the number of processors is more than eight, the working set can be accommodated in the cache under the CAS scheme, so CAS outperforms FSI and SCTV. For the size of 256, the working set is larger than the cache, thus all the schemes except No Cache obtain similar speedups.

CG. In CG, the sparse matrix is read-only, so this matrix is cached without the coherence problem in all the versions including No Cache. However, the No Cache version still performs poorly (see Figure 5.12). Although FSI and SCTV retain only spatial locality in this application, they perform much better than the No Cache version. The CAS scheme has speedups close to the upper bound given by the No Coherence version for all numbers of processors.

5.6 Summary

In this chapter, we have proposed the Cache Affinity-based Software cache coherence scheme. This scheme exploits data locality across parallel loops, but it does not require modifications to the cache memory hardware. We have proposed parametric sections to represent cache affinity. We have developed detailed algorithms for computing cache affinity among parallel loops and inserting *invalidate* and *update* operations for cache coherence.

CAS has been evaluated on the Hector multiprocessor by comparing it with other schemes. The experi-

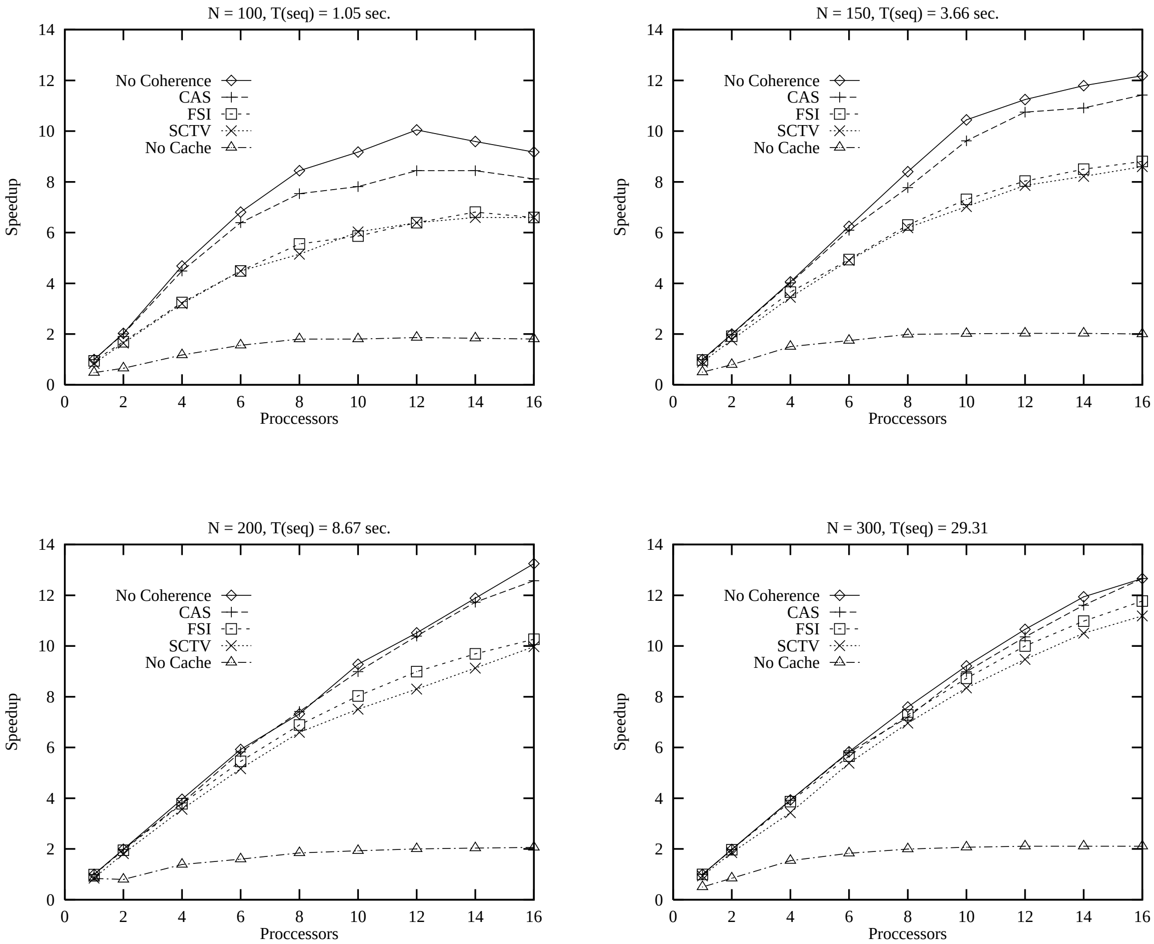
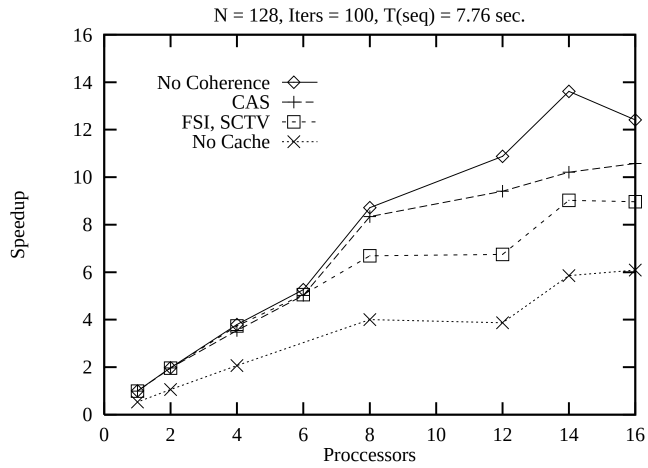


Figure 5.10: Speedups of LU Decomposition



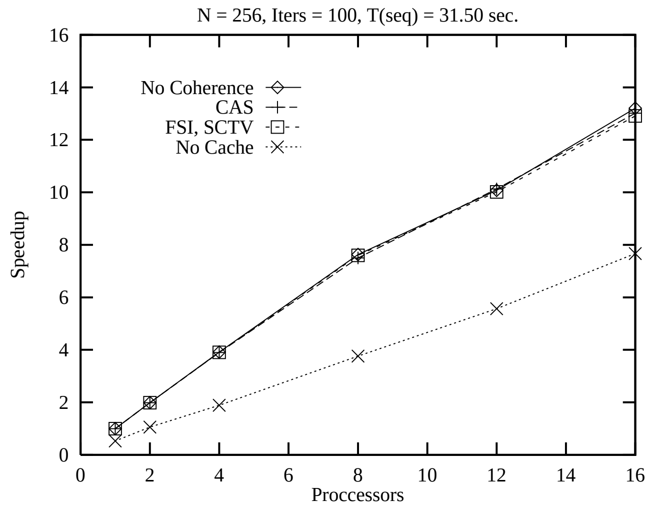


Figure 5.11: Speedups of SOR

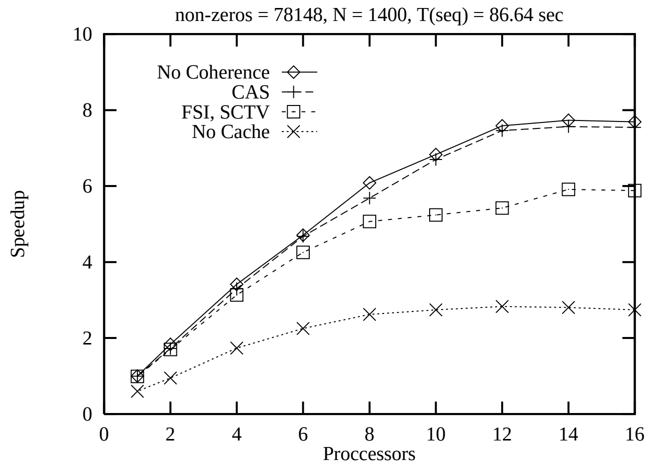


Figure 5.12: Speedups of CG

mental results have shown that:

- Caching data is very important to performance. All three applications studied perform poorly without using caches.
- With a small cache size, simple software cache coherence schemes perform well since data reuse in small caches is dominated by spatial locality, which can be easily achieved by these simple schemes.
- When a large portion of the working set can be accommodated in the cache, temporal locality affects performance significantly. CAS exploits such cache affinity and achieves higher cache hit ratios, thus outperforming other schemes.

Since regular sections have been used in interprocedure analysis and data distribution analysis in ParaScope [HK91], we can replace regular sections by parametric sections. The precise set operations for regular sections should also be changed to the approximate set operations. Then the algorithms for computing cache affinity can be implemented based on those approximate set operations. In order to reduce cache line false sharing, it is necessary to apply data transformations before CAS. The rules discussed in Section 5.4 can be used to derive such data transformations.

Chapter 6

Conclusion

6.1 Summary of the Thesis

On scalable shared-memory multiprocessors, parallel programming is a challenging task. Without fine tuning, parallel programs designed for UMA machines suffer from excessive remote accesses and deliver poor speedups. Hand-tuning the code can be quite difficult and may require detailed knowledge of machine characteristics. Moreover, the code well-tuned for one particular machine is not portable to other machines without a high likelihood that good performance would be lost. To ease the burden on the programmer and to achieve portability, software support for reducing memory access latency should be provided by compilers and run-time systems.

While loop transformations can improve data locality, data layout in memory and cache also impacts data locality more directly. To complement previous research, which focuses on loop transformation theory, we have directed our attention to data management. We have proposed three types of software techniques: data transformations for good data layout to avoid remote accesses, new load balancing schemes for both data locality and load balancing, and a new software cache coherence scheme for reducing the number of coherence cache misses.

With the software techniques proposed in Chapters 2 to 5, programmers only need to provide high-level information such as choosing between block and cyclic data distributions without considering operating systems, cache coherence, and underlying architectures. These techniques reduce the number of remote memory accesses and the number of cache misses, and therefore the average memory access time is kept low. As a result, parallel applications with such software support can achieve good speedups on scalable multiprocessors. The trends of importance in planning for future multiprocessors are: (1) processor speeds are increasing faster than the memory speeds; (2) cache sizes are becoming larger; and (3) remote memory access times are also increasing with the number of processors. Therefore, we expect that the software techniques developed in this thesis for reducing memory latency and improving data locality in memory and cache will be even more important to future multiprocessors.

6.1.1 Data Transformations

To complement the various loop transformations, we have proposed a data transformation framework that enriches the range of data layout schemes in multiprocessors. Loop transformations are constrained by data dependencies, but data transformations are always applicable. With a comprehensive set of data transformations, we can choose the best data layout for arrays in each parallel application. We have applied data transformations to enhance and implement data distributions, to reduce communication, and to improve cache locality.

Data Distributions. In the data transformation framework, we have combined regular data distributions with unimodular transformations to form a rich set of data distributions. The regular data distributions in many parallel language extensions are a subset of data transformations. Transformed arrays are used to implement the data distributions for the original arrays. The transformed arrays not only reduce mismatching and false sharing remote memory accesses, but also simplify code generation by using the same array names for all processors. The NUMACROS macro package is a prototype that demonstrates the feasibility of data distribution coordinated with loop iteration scheduling. We have shown that implementing data parallel programs on scalable shared-memory machines can be made much easier than on message-passing machines.

Improving Cache Locality. Data layout in main memory also impacts cache performance. For example, if an array larger than the cache is stored in row-major order but accesses are made in column-major order, the spatial locality in the cache line cannot be exploited. Unimodular data transformations can be used for achieving the best order of array dimensions. We have developed a global optimization algorithm to determine (1) what array dimension order maximizes data locality in each parallel loop, and (2) in which parallel loops data transformations need to be applied. We have shown that such techniques significantly reduce the execution times in four of the seven NAS kernel programs.

Reducing Communication. Since a cache line can hold several words, remote accesses can be reduced by mapping individual words in different cache lines into the same cache line. To achieve this, we have defined several data transformations such as partitioning, scaling, and fusion, and shown significant performance gains in several parallel applications.

Optimizing Index Computation. Despite the closed forms for computing indices to access transformed arrays, they involve expensive operations such as modulus, multiplication, and division. Directly using the closed form in indexing can make index computation itself so expensive that speedups are nullified in most programs. Current compilers do not optimize such operations in index computation. We propose several optimization techniques that effectively make such index computation overhead negligible.

The data transformations can be incorporated into compilers after the parallelizing phase as follows. The techniques discussed in Chapter 2 for implementing parallel loops and data distributions can be directly used in compilers. Cache locality optimization by permuting array dimension orders should be applied after data transformations for data distributions and optimizing communication. The methods for indexing computations should be implemented after data transformation phases, and both the strength reduction and loop transformation for indexing should be used.

6.1.2 Locality and Load Balancing

For parallel applications in which the computation times of loop iterations are highly variable or unpredictable, both data locality and load balancing are important to performance. Most load balancing schemes do not take data distribution into account, so they can destroy the data locality supplied by our data transformations.

Locality-based Dynamic Scheduling (LDS). This scheme is useful for scheduling parallel loops in which iteration execution times are highly variable and unpredictable. It uses data distribution information to retain data locality and applies global dynamic loop scheduling schemes to achieve good load balance. The global dynamic scheduling scheme in LDS uses few lock operations and avoids having to access many remote memories in order to identify the set of subtasks that should next be assigned to a processor. We have experimentally compared LDS with other loop scheduling schemes in different types of applications. LDS uniformly performs as well as the best of other dynamic scheduling schemes across a variety of applications.

Overpartitioning. In many applications (for example parallel sorting), the amount of computation can be determined by the data space size, but it is difficult to partition the global space into equal-sized subspaces for each processor. We have proposed an overpartitioning technique to deal with such load imbalance. We have employed this technique in both comparison based and non-comparison based parallel sorts, resulting in two efficient algorithms: Parallel Quicksort by OverPartitioning (PQOP) and Parallel Radixsort by OverPartitioning (PROP). We have shown that overpartitioning is a very effective technique and leads to good load balancing with high probability. Parallel sorting by overpartitioning also achieves good data locality because it assures that each array element is moved from one processor's physical memory module to another's at most once. We have demonstrated that both PQOP and PROP achieve nearly linear speedups and outperform other parallel sorting algorithms of which we are aware on scalable shared-memory multiprocessors.

Both LDS and overpartitioning can be incorporated into run-time systems. The compiler should determine whether there is load imbalance in the code and of which type the load imbalance is, then choose an appropriate load balancing technique. LDS can be built into run-time libraries for scheduling parallel loops. Overpartitioning requires programmers to apply particular partitioning methods to their applications, but the building of the task queues and the scheduling of the tasks can be incorporated into run-time systems as well.

6.1.3 Cache Affinity-based Software Cache Coherence Scheme

As cache sizes increase, cache affinity becomes more important to performance. Software cache coherence is less expensive than directory-based hardware cache coherence, but most current software schemes ignore cache affinity across parallel loops. We develop a Cache Affinity-based Software scheme (CAS) that allows valid data to reside in caches across parallel loops, resulting in higher cache hit ratios.

Reducing coherence cache misses. In each parallel loop, the read cache affinity corresponds to the data that is not modified by other processors, and the write cache affinity corresponds to the data that will be written again by the same processor. In the CAS scheme, read cache affinity data is not invalidated in the cache, and write cache affinity data is not updated into main memory, so the number of coherence cache misses is reduced. We have proposed parametric sections to specify cache affinity. We have developed the detailed algorithms for computing cache affinity across parallel loops and inserting the minimal *invalidate* and *update* operations required for cache coherence.

Avoiding cache line false sharing. Because of cache line false sharing, existing software schemes use a write-through policy for caches so that each write immediately updates main memory. To avoid cache line false sharing, we adjust array sizes to be cache line aligned, and partition loop iterations to reduce write-write false sharing, where two or more processors update the same cache line.

We have evaluated CAS and compared it with a few other cache coherence schemes on Hector. The results have shown that CAS outperforms other software schemes and is competitive with schemes that require hardware support in the applications we investigated.

Software cache coherence is the last phase in parallelizing compilers for NUMA multiprocessors without hardware support. In Chapter 5, we have proposed detailed algorithms to compute data sets for invalidating and updating cache. These algorithms can directly be implemented in compilers.

While the software techniques proposed in this thesis are directly applicable for scalable shared-memory multiprocessors, they can also be extended to other types of machines

- *Uniprocessors.* With increasing cache sizes, data locality in the cache has a greater impact on performance. The data transformations and algorithms for deriving data transformations proposed in Chapter 3 can be applied directly to cache optimization for uniprocessors.
- *Small-scale shared-memory multiprocessors.* Despite the uniform access time to main memory, cache affinity is critical to performance on this type of multiprocessors. Therefore, the data transformations that improve data locality on scalable shared-memory machines can be used to improve cache affinity on small-scale shared-memory machines with large caches. LDS can also be directly used to retain cache affinity. PSOP can achieve nearly linear speedups. Although cache coherence is enforced by hardware on small-scale shared-memory machines, the algorithms for computing cache affinity in CAS can be used to statically schedule parallel loops for retaining cache affinity across them.
- *Message-passing multiprocessors.* On message-passing multiprocessors or workstation clusters, data distributions can be implemented to facilitate data parallel programming as well. Data transformations for reducing communication and improving cache locality are also important and applicable. LDS can be implemented in a similar way, except that the task queue should be handled differently. PSOP can perform well on message-passing machines with some modifications to the implementation of load balancing. Finally, the algorithms for computing cache affinity in CAS can be used to reduce redundant communications.

6.2 Future Work

We briefly outline some topics of future research that are related to the thesis.

- **Interaction of data transformations with loop transformations.**

  Loop transformations can improve data locality, but they are constrained by data dependences. Data transformations can always be applied for improving locality, however they may require the permutation of arrays. Therefore, it is necessary to combine the two types of transformations to increase data locality while reducing the cost for permutation. A possible approach is to propose a framework to unify these transformations and develop algorithms to generate the transformations for parallel applications.

- **Shared-memory for message-passing multiprocessors.**

  Current message-passing multiprocessors such as IBM SP/2 support direct mapping of a portion of the memory on one processor to a portion of the memory on the other processor. With such hardware support, we will be able to provide the shared-memory paradigm by software. The CAS cache coherence scheme can be extended for supporting shared-memory on message-passing machines in fine granularity. While the CAS scheme is a purely compile-time approach to handling regular data structures and parallel loops, a run-time approach could be developed to handle irregular and dynamic data structures.

- **Overpartitioning in other applications.**

  While overpartitioning has been successfully applied only to parallel sorting, it may also be useful in other applications. To directly use the overpartitioning technique, the applications must be partitioned into relatively independent tasks for which the execution times can be estimated.

- **Locality and scheduling in multiprogrammed multiprocessors**

  In this thesis, we focus on data locality and load balancing for single applications in multiprocessors. In multiprogramming environments, memory management, and processor allocation and scheduling should take application characteristics into account [Sev94]. Data locality and sharing may not be limited to sharing data in main memory within single applications. Examples of other types of data locality are sharing data buffers and files among applications. Further research should develop a scheduling theory that determines how to make trade-offs between these types of data locality and scheduling policies, and provide practical solutions for a better multiprogramming environment on multiprocessors.

Bibliography

[AAHV91] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon. Comparison of Hardware and Software Cache Coherence Schemes. In *Proc. of the 18th Annual International Symposium on Computer Architecture*, pages 298–308, May 1991.

[ACJ+91] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Architectures*. Kluwer Academic Publishers, 1991.

[AK87] J. R. Allen and K. Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):617–640, 1987.

[AL93] J. M. Anderson and M. S. Lam. Global Optimization for Parallelism and Locality on Scalable Parallel Machines. In *Proc. of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 112–125, June 1993.

[Alv90] G. A. Alverson. Abstractions for Effectively Portable Shared Memory Parallel Programs. Technical Report 1990-10-09, Department of Computer Science and Engineering, University of Washington, Seattle, WA, October 1990.

[Amd67] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proc. of AFIPS*, pages 483–485, 1967.

[AN91] G. A. Alverson and D. Notkin. Abstracting Data-Representation and Partition-Scheduling in Parallel Programs. In *Proc. of International Symposium on Shared Memory Multiprocessing*, pages 138–151, Toyko, Japan, April 1991.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[Bal90] V. Balasundaram. A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

[Bat68] K. Batcher. Sorting Networks and Their Applications. In *Proc. of AFIPS*, pages 307–314, 1968.

[BBM+89] M. Beltrametti, K. Bobey, R. Manson, M. Walker, and D. Wilson. PAMS/SP-2 System Overview. In *Proc. of Supercomputing Symposium*, pages 63–71, 1989.

[BBN90] BBN Advanced Computers, Inc. *Inside the TC2000 Computer*, 1990.

[BCH+93] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a Portable Nested Data-Parallel Language. In *Proc. of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, CA, May 1993.

[Bel92] G. Bell. Ultracomputers: A Teraflop Before Its Time. *Communications of ACM*, 35(8):27–47, August 1992.

[BFS89] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proc. of 12th ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[BGW89] D. Black, A. Gupta, and W. Weber. Competitive Management of Distributed Shared Memory. In *Proc. Spring Compcon*, pages 184–190, San Francisco, CA, February 1989.

[BH85] A. Borodin and J. Hopcroft. Routing, Merging and Sorting on Parallel Models of Computation. *Journal of Computer Systems and Science*, 30:130–145, 1985.

[Ble90] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[BLL88] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software Practice and Experience*, 18(8):713–732, August 1988.

[BLM+91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proc. of 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'91)*, pages 3–16, Hilton Head, SC, July 1991.

[BR93] D. T. Blackston and A. Ranade. SnakeSort: A Family of Simple Optimal Randomized Sorting Algorithms. In *Proc. of 22nd International Conference on Parallel Processing*, volume III: Applications, pages 201–204, St. Charles, IL, August 1993.

[BSF+91] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proc. of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[Car93] S. Carr. *Memory-Hierarchy Management*. PhD thesis, Department of Computer Science, Rice University, Houston, TX, February 1993.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 152–64, October 1991.

[CF89] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experience with PLATINUM. In *Proc. of 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[CGH93] R. Chandra, A. Gupta, and J. L. Hennessy. Data Locality and Load Balancing in COOL. In *Proc. of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–259, May 1993.

[Che92] H. Cheong. Lift Span Strategy – A Compiler-Based Approach to Cache Coherence. In *Proc. of 5th International Conference on Supercomputing*, July 1992.

[CKM88] R. Cytron, S. Karlovsky, and K. P. McAulife. Automatic Management of Programmable Caches (Extended Abstract). In *Proc. of 17th International Conference on Parallel Processing*, pages 229–238, St. Charles, IL, August 1988.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proc. of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991.

[CL94] M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared Memory Machines. Technical Report 542, Department of Computer Science, University of Rochester, Rochester, NY, November 1994.

[Col88] R. Cole. Parallel Merge Sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[CP90] D. Callahan and A. Porterfield. Data Cache Performance of Supercomputer Applications. In *Proc. of 3th International Conference on Supercomputing*, June 1990.

[CRI92] Cray T3D Software Overview. Technical Note, Cray Research, Inc, Eagan, Minnesota 55121. 1992.

[CS90] D. Callahan and B. Smith. A Future-based Parallel Language for a General-purpose Highly Parallel Computer. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 95–113. Pitman, London, 1990.

[CV88a] H. Cheong and A. Veidenbaum. Cache Coherence Schemes with Fast Selective Invalidation. In *Proc. of the 15th Annual International Symposium on Computer Architecture*, pages 299–307, May 1988.

[CV88b] H. Cheong and A. Veidenbaum. Stale Data Detection and Coherence Enforcement Using Flow Analysis. In *Proc. of 17th International Conference on Parallel Processing*, pages 138–145, St. Charles, IL, August 1988.

[CV90] H. Cheong and A. Veidenbaum. Compiler-Directed Cache Management in Multiprocessors. *Computer*, 23(6):39–47, 1990.

[CV91] Y. Chen and A. V. Veidenbaum. Comparison and Analysis of Software and Directory Coherence Schemes. In *Proc. of Supercomputing'91*, pages 818–829, 1991.

[CV92] Y. Chen and A. V. Veidenbaum. An Effective Write Policy for Software Coherence Schemes. In *Proc. of Supercomputing'91*, pages 661–672, 1992.

[DMCK92] E. Darnell, J. M. Mellor-Crummey, and K. Kennedy. Automatic Software Cache Coherence through Vectorization. In *Proc. of 5th International Conference on Supercomputing*, July 1992.

[FHK+90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical Report TR90-141, Department of Computer Science, Rice University, Houston, TX, December 1990.

[FJL+88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume I: General Techniques and Regular Problems. Prentice Hall, 1988.

[FP92] R. S. Francis and L. J. H. Pannan. A Parallel Partition for Enhanced Parallel Quicksort. *Parallel Computing*, 18:543–550, 1992.

[GB92] M. Gupta and P. Banerjee. Automatic Data Partitioning on Distributed Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[Gep95] L. Geppert. Solid State. *IEEE Spectrum*, 32(1):35–39, January 1995.

[GGK+83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer. In *Proc. of the 9th Annual International Symposium on Computer Architecture*, pages 27–42, May 1983.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, NY, 1979.

[GJG88] D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[GV91] E. D. Granston and A. V. Veidenbaum. Detecting Redundant Accesses to Array Data. In *Proc. of Supercomputing'91*, pages 854–865, 1991.

[HA90] D. Hudak and S. Abraham. Compiler Techniques for Data Partitioning of Sequentially Iterated Parallel Loops. In *Proc. of 3th International Conference on Supercomputing*, June 1990.

[HC83] J. S. Huang and Y. C. Chow. Parallel Sorting and Data Partitioning by Sampling. In *Proc. of the IEEE Computer Society's 7th International Computer Software and Applications Conference*, pages 627–631, 1983.

[HK91] P. Havlak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. In *Proc. of Supercomputing'91*, pages 86–100, Albuquerque, NM, November 1991.

[HKT92] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed Machines. In *Proc. of 5th International Conference on Supercomputing*, pages 1–14, July 1992.

[HLJ+91] P. J. Hatcher, A. J. Lapadula, R. R. Jones, M. J. Quinn, and R. J. Anderson. A Production-Quality C* Compiler for a Hypercube Multicomputer. In *Proc. of 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[HP90] J. L. Hennessey and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[HPF93] High Performance Fortran Language Specification (High Performance Fortran Forum). Technical Report Draft, Rice University, Houston, TX, January 1993.

[HPR92] W. L. Hightower, J. F. Prins, and J. H. Reif. Implementation of Randomized Sorting on Large Parallel Machines. In *Proc. of 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'93)*, pages 158–167, San Diego, CA, July 1992.

[HQA+91] P. J. Hatcher, M. J. Quinn, R. J. Anderson, A. J. Lapadula, B. K. Seevers, and A. F. Bennett. Architecture-Independent Scientific Programming in Dataparallel C: Three Case Studies. In *Proc. of Supercomputing'91*, pages 208–217, Albuquerque, NM, November 1991.

[HS86] W. D. Hills and G. L. Steele Jr. Data Parallel Algorithms. *Communications of ACM*, 29(12):1170–1183, December 1986.

[HSF92] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Communications of ACM*, 35(8):90–101, August 1992.

[IT88] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proc. of the 15th ACM Symposium on the Principles of Programming Languages*, January 1988.

[KK93] L. V. Kalé and S. Krishnan. A Comparison Based Parallel Sorting Algorithm. In *Proc. of 22nd International Conference on Parallel Processing*, volume III: Applications, pages 196–200, St. Charles, IL, August 1993.

[KM91] C. Koelbel and P. Mehrotra. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2:440–451, October 1991.

[KM92] K. Kennedy and K. McKinley. Optimizing for Parallelism and Memory Hierarchy. In *Proc. of 5th International Conference on Supercomputing*, pages 223–234, July 1992.

[KMV90] C. Koelbel, P. Mehrotra, and J. V. Rosendale. Supporting Shared Data Structures on Distributed Memory Machines. In *Proc. of 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[KSR91] Kendall Square Research. *KSR1 Principles of Operation.* Waltham, MA, 1991.

[KSR93] Kendall Square Research. *KSR Fortran Programming.* Waltham, MA, 1993.

[KTV+91] J. Konicek, T. Tilton, A. Veidenbaum, C. . Zhu, E. S. Davidson, R. Downing, M. Haney, M. Sharma, P. C. Yew, P. M. Farmwald, D. Kuck, D. Lavery, R. Lindsey, D. Pointer, J. Andrews, T. Beck, T. Murphy, S. Turner, and N. Warter. The Organization of the Cedar System. In *Proc. of 20th International Conference on Parallel Processing*, volume I: Hardware, pages 49–56, St. Charles, IL, August 1991.

[KW85] C. P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, October 1985.

[LC90] J. Li and M. Chen. Synthesis of Explicit Communication from Shared-Memory Program References. In *Proc. of Supercomputing'90*, New York, NY, November 1990.

[LC91] J. Li and M. Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *Journal of Parallel and Distributed Computing*, 2(3):361–376, July 1991.

[LE90] R. P. LaRowe Jr. and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. Technical Report CS-1990-10, Department of Computer Science, Duke University, Durham, NC, April 1990.

[Lei85] F. T. Leighton. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Transactions on Computers*, C–34(4):344–354, April 1985.

[LEK91] R. P. LaRowe Jr., C. S. Ellis, and L. S. Kaplan. Tuning NUMA Memory Management for Applications and Architectures. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pages 152–64, October 1991.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Department of Computer Science, Yale University, September 1986.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LLS+93] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19(10):543–550, October 1993.

[LRW91] M. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, April 1991.

[LS93] H. Li and K. C. Sevcik. NUMACROS: Data Parallel Programming on NUMA Multiprocessors. In *Proc. of 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 247–263, San Diego, CA, September 1993.

[LS94a] H. Li and K. C. Sevcik. Exploiting Cache Affinity in Software Cache Coherence. In *Proc. of 7th International Conference on Supercomputing*, pages 264–273, Manchester, England, UK, July 1994.

[LS94b] H. Li and K. C. Sevcik. Parallel Sorting by Overpartitioning. In *Proc. of 6th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'94)*, pages 46–56, Cape May, NJ, June 1994.

[LT90] P. P. Li and Y. Tung. Parallel Sorting on Symult 2010. In *Proc. of 5th Distributed Memory Computing Conference*, pages 224–229, Charleston, SC, April 1990.

[LTSS93] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and Loop Scheduling on NUMA Multiprocessors. In *Proc. of 22nd International Conference on Parallel Processing*, volume II: Software, pages 140–147, St. Charles, IL, August 1993.

[LWE91] R. P. LaRowe Jr., J. T. Wilkes, and C. S. Ellis. Exploiting Operating System Support for Dynamic Page Placement on a NUMA Shared Memory Multiprocessor. In *Proc. of 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[LY91] D. J. Lilja and P. Yew. Combining Hardware and Software Cache Coherence Strategies. In *Proc. of 4th International Conference on Supercomputing*, July 1991.

[LZ93] S. Leung and J. Zahorjan. Improving Performance of Runtime Parallelization. In *Proc. of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–259, May 1993.

[LZGS84] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[MB89] S. L. Min and J. Baer. A Timestamp-based Cache Coherence Scheme. In *Proc. of 18th International Conference on Parallel Processing*, volume I: Hardware, pages 23–32, St. Charles, IL, August 1989.

[MB90] S. L. Min and J. Baer. A Performance Comparison of Directory-based and Timestamp-Based Cache Coherence Schemes. In *Proc. of 19th International Conference on Parallel Processing*, volume I: Hardware, pages 305–311, St. Charles, IL, August 1990.

[ML94] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 379-400, 5(4): April 1994.

[MLG92] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proc. of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.

[NS82] D. Nassimi and S. Sahni. Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network. *Journal of the ACM*, 29(3):642–667, July 1982.

[OA89] S. Owicki and A. Agarwal. Evaluating the Performance of Software Cache Coherence. In *Proc. of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 230–242, 1989.

[PK87] C. Polychronopoulos and D. Kuck. Guided Self Scheduling: A Practical Scheduling Scheme for Parallel Computers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[Pla89] C. G. Plaxton. Efficient Computation on Sparse Interconnection Networks. Technical Report STAN-CS-89-1283, Department of Computer Science, Stanford University, Stanford, CA, September 1989.

[Por89] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Department of Computer Science, Rice University, Houston, TX, May 1989.

[PS85] J. L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, Reading, MA, 1985.

[QH91] M. J. Quinn and P. J. Hatcher. Data Parallel Programming on Multicomputers. *IEEE Software*, 7:69–76, September 1991.

[QHS91] M. J. Quinn, P. Hatcher, and B. Seevers. Implementing a Data Parallel Language on a Tightly Coupled Multiprocessor. In Alexander Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Advances in Languages and Compilers for Parallel Computing*, pages 385–401. Pitman, London, 1991.

[Qui89] M. J. Quinn. Analysis and Benchmarking of Two Parallel Sorting Algorithms: Hyperquicksort and Quickmerge. *BIT*, 29(2):239–250, 1989.

[RS87] J. R. Rose and G. L. Steele Jr. C*: An Extended C Language for Data Parallel Programming. Technical Report TR. PL 87–5, Thinking Machines Corporation, Boston, MA, 1987.

[RS91] J. Ramanujam and P. Sadayappan. Compile Time Techniques for Data Distribution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[RSG93] E. Rothber, J. P. Singh, and A. Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[RSW90] M. Rosing, R. Schnabel, and R. Weaver. The DINO Parallel Programming Language. Technical Report CU-CS-457-90, University of Colorado, April 1990.

[RV87] J. H. Reif and L. G. Valiant. A Logarithmic Time Sort for Linear Size Networks. *Journal of the ACM*, 34(1):60–76, January 1987.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-Time Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[Sev94] K. C. Sevcik. Application Scheduling and Processor Allocation. *Performance Evaluation*, 19(2–3):107–140, March 1994.

[SGZ93] H. S. Sandhu, B. Gamsa, and S. Zhou. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Proc. of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–238, May 1993.

[SIK+94] T. Shindo, H. Iwashita, S. Kaneshira, T. Doi, and J. Hagiwara. Twisted Data Layout. In *Proc. of 7th International Conference on Supercomputing*, pages 374–381, Manchester, England, UK, July 1994.

[SS92] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:362–372, 1992.

[Ste90] P. Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23(6):12–24, 1990.

[TB93] A. Tridgell and R. P. Brent. An Implementation of a General-Purpose Parallel Sorting Algorithm. Technical Report TR-CS-93-01, Computer Science Laboratory, Australian National University, Australia, February 1993.

[TN91] T. H. Tzen and L. M. Ni. Dynamic Loop Scheduling for Shared Memory Multiprocessors. In *Proc. of 20th International Conference on Parallel Processing*, volume II: Software, pages 247–250, St. Charles, IL, August 1991.

[TS92] K. Thearling and S. Smith. An Improved Supercomputer Sorting Benchmark. In *Proc. of Supercomputing'92*, pages 307–314, November 1992.

[Unr93] R. C. Unrau. *Scalable Memory Management through Hierarchical Symmetric Multiprocessing.* PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada, 1993.

[VSWL91] Z. G. Vranesic, M. Stumm, R. White, and D. Lewis. The Hector Multiprocessor. *Computer*, 24(1), January 1991.

[Wag90] B. A. Wagar. *Practical Sorting Algorithms for Hypercube computers.* PhD thesis, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, July 1990.

[WL91] M. E. Wolf and M. S. Lam. A Data Locality Optimization Algorithm. In *Proc. of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[Wol87] M. Wolfe. Iteration Space Tiling for Memory Hierarchies. In *Proc. of the 3rd SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

[Wol89a] M. Wolfe. More Iteration Space Tiling. In *Proc. of Supercomputing'90*, 1989.

[Wol89b] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, 1989.

[Wol91] M. Wolfe. The Tiny Loop Restructuring Research Tool. In *Proc. of 20th International Conference on Parallel Processing*, volume II: Software, pages 47–53, St. Charles, IL, August 1991.

[Wol92] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, August 1992.

[ZBG86] H. Zima, H. Bast, and M. Gerndt. Superb: A Tool For Semi-automatic MIMD/SIMD Parallelization. *Parallel Computing*, 6:1–18, 1986.

[ZSLW92] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, September 1992.

Appendix A

NUMACROS for the Hector Multiprocessor

This appendix gives a part of the NUMACROS implementation for the Hector multiprocessors. The file "numacros.h" contains

- Hector dependent code for synchronization, processor creation, and timing
- Parallel loops and data distributions on 1-D grid of processors. Data distributions use the precomputing approach to index computation.

The small file "distribution.m" is m4 macros for data distributions using the naive approach to index computation. Note that m4 can be used to define data distributions for 2-D grid of processors. The file "main.c" starts parallel programs in NUMACROS.

```
/*************************************************************************
 *              NUMACROS: macros for Hector NUMA Multiprocessor          *
 *              numacros.h                                               *
 *************************************************************************/

#include <stdio.h>
#include <sys/time.h>

/************** standard .h files *********/
#ifdef HURRI
#include <hurricane.h>
#include <hurricane/memory.h>
#include <hurricane/lock.h>
#endif

/************** Consts ********************/
#ifndef MaxProcs
#define MaxProcs        16
#endif
#ifndef PageSize
#define PageSize        4096
#endif

#ifndef Pages
#define Pages           8
#endif
#ifndef CacheSize
#define CacheSize       1024*16
```

```
#endif
#ifndef min
#define min(a, b)       (((a)<(b))?(a):(b))
#endif
#ifndef max
#define max(a, b)       (((a)>(b))?(a):(b))
#endif

/*************** System Dependent Types/Macros ************/
#ifdef HURRI
      /* hurricane version */

#define SystemCode_type SystemCode
#define slock_type      slock
#define hlock_type hlock
#define barrier_type    barrier
#define ProcessId_type  ProcessId

#define __BindRegion(size, perm) BindRegion(0,-1,0,0, size, perm, &rc)

#ifndef MALLOC
#define MALLOC(size)     __BindRegion(size, BIND_WEAK| BIND_FIRST_HIT)
#endif

#define __InitBarrier(bar, numprocs) InitBarrier(bar, numprocs)
#define __InitSpinLock(lock) InitSpinLock(lock)
#define __InitLock(lock) InitLock(lock)
#define __SpinLock(lock) SpinLock(lock)
#define __SpinUnlock(lock) SpinUnlock(lock)
#define __XMEM XMEM
#define __SpinOnlyBarrier(bar)          SpinOnlyBarrier(bar)

#define __FlushCache(p, size) FlushCache(p, size)

#else /* sequential version */

#define SystemCode_type int
#define slock_type      int
#define hlock_type int
#define barrier_type    int
#define ProcessId_type int

#define MALLOC(size)    malloc(size)

#define __InitBarrier(bar, numprocs)
#define __InitSpinLock(lock)
#define __InitLock(lock)
#define __SpinLock(lock)
#define __SpinUnlock(lock)
#define __SpinOnlyBarrier(bar)
#define __XMEM(r, lock)  (___tmp = lock, lock = r, lock)
#define __FlushCache(p, size)

#endif /* HURRI */

/********************* timers **********************************/
/* timers are also system-dependent.  */
```

```
#ifdef   uTIMER
             /* micro second timers */
#define  Time_type       unsigned
#define  Get_Timer(time) {time = Getutimer()};
#define  lstart_timer()  __lstart_time = Getutimer()
#define  lstop_timer()   __lend_time = Getutimer()
#define  l_timing()      (__lend_time - __lstart_time)
#define  lprint_timer(s) printf("On proc %d is %d ticks %s\n",\
                                myPid, l_timing(),s)
#else

#define  Time_type        struct timeval
#define  Get_Timer(time) {gettimeofday(&(time), 0);}
#define  lstart_timer()   gettimeofday(&__lstart_time, 0)
#define  lstop_timer()    gettimeofday(&__lend_time, 0)
#define  l_timing() ((int)((1000.0*(__lend_time.tv_sec-__lstart_time.tv_sec)) \
                     + ((__lend_time.tv_usec-__lstart_time.tv_usec)/1000.0)))
#define lprint_timer(s) printf("On proc %d is %d mili seconds %s\n", \
                               myPid, l_timing(), s)

#endif /* uTIMER */

/******************* the context of process ***************/

/* Each process needs a copy of necessary shared variables in
   its stack, because Hurricane may make globally shared
   variables uncacheable in most time
*/

#define SharedVars      slock_type      __lock_var;     \
                        barrier_type    __reduce_bar; \
                        barrier_type    __bar;          \
                        int             __numprocs = MaxProcs;  \
                        int             __timing = 1;   \
                        int             __millis = 0;   \
                        int             __pages;        \
                        int             __p[MaxProcs]; \
                        slock_type      __locks[MaxProcs]; \
                        int             __SchedLock=0; \
                        char            * __SMem

/* The global copy of the shared variables */
#define ExtSharedVars   extern  slock_type      __lock_var;     \
                        extern  barrier_type    __reduce_bar; \
                        extern  barrier_type    __bar;          \
extern  int __numprocs;     \
                        extern  int             __timing;       \
                        extern  int             __millis;       \
                        extern  int             __pages;        \
                        extern  int             __SchedLock; \
                        extern  int             __p[MaxProcs]; \
                        extern  slock_type      __locks[MaxProcs]; \
                        extern  char            * __SMem

/* These are the necessary varibles for each process */

#define PerProcessVars unsigned int * __cacheP; \
                register unsigned myPid;    \
```

```
                register unsigned PROCS;        \
                Time_type __lstart_time, __lend_time; \
                LSharedMemory __LMem; \
                SystemCode_type rc

/* __cacheP is used for flushing the entire cache. */
#define INIT_CACHEP { __cacheP = (unsigned *) __LMem.start; }

/* Initially, each processor maps a portion of shared-memory
   address space into its local physical memory. Later, mapping
   macros are used for distributing arrays among processors, i.e,
   allocating memory from this.
*/

#define INIT_LMem{      char * p; int i; \
        __LMem.start  = (void*) MALLOC((__pages+1)*PageSize); \
        if (__LMem.start ==NULL) \
            { printf("no mem for shared \n"); return;} \
        __LMem.next = __LMem.start; __LMem.left = __pages*PageSize; \
        for (i = 0, p = (char*) __LMem.start; i <__pages; i++) \
                p[i*1024] = 0; \
}

#define InitPerProcessVars { INIT_LMem;  INIT_CACHEP;\
                              myPid = myPid; PROCS = __numprocs ;}

/************** mem mapping ***************/

typedef struct {
        void * start;
        void * next;
        unsigned left;
} LSharedMemory;

/* Make len be cache line alignment */
#define allocSize(len)  ( (len & 0x07) ? ((((len)>>4)<<4) + 0x10): (len))

#define lmalloc(len) (( (__LMem.left  -= (len)) && (__LMem.next += (len)))? \
                        (__LMem.next - (len)): NULL)

#define Partition(A, B) ((A)/(B) + ((A%B) != 0))

/*******************************************************************
   row-oriented mapping:
        A:  a 1-dimensional array of pointers to rows
        GL: global pointer, LA, local pointer.
*******************************************************************/

#define __BlockMap__(i, rows, part)     \
        for (i = myPid*part; i <min((myPid+1)*part, rows); i++){

#define __CyclicMap__(i, rows, part)    \
        for (i = myPid; i <rows; i += PROCS){

#define __Blk_CyclicMap__(i, rows, brows)    \
for (i = 0; i< rows; i++) \
                if (myPid == (i/brows % PROCS)) {
```

```
#define __Mapping__(GA, LA, rows, cols, brows, Type, MAP) \
      { char *p; int i, part; \
if ((LA = (Type**) lmalloc(allocSize(sizeof(Type*)*rows))) == NULL){ \
printf("LA Out of memory on processor %d \n", myPid);\
}\
        part = Partition(rows, PROCS); \
        if ( (p = lmalloc(allocSize(part*sizeof(Type)*cols))) == NULL) {\
                printf("Out of memory on processor %d \n", myPid);\
}\
        MAP(i, rows, part)      \
             GA[i] = (Type*) p; p += (cols*sizeof(Type)); \
}       \
        __BARRIER;\
        for (i = 0; i<rows; i++) LA[i] = GA[i]; \
      }


#define blk_map(GA, LA, rows, cols, Type)  \
         __Mapping__(GA, LA, rows, cols, 1,  Type, __BlockMap__)

#define cyc_map(GA, LA, rows, cols, Type)  \
         __Mapping__(GA, LA, rows, cols, 1, Type, __CyclicMap__)

#define blk_cyc_map(GA, LA, rows, cols, brows, Type) \
          __Mapping__(GA, LA, rows, cols, brows,  Type, __Blk_CyclicMap__)

/********************** basic syn *****************/
#define __LOCK()        __SpinLock(&__lock_var)
#define __UNLOCK()      __SpinUnlock(&__lock_var)
#define __BARRIER __SpinOnlyBarrier(&__bar)
#define BARRIER         __BARRIER
#define __FLUSH_ALL     __FlushCache(__cacheP, 1024*32+4)
#define FLUSH_ALL       __FlushCache(__cacheP, 1024*32+4)
#define __DataLock      __SpinLock(&__reduce_lock)
#define __DataUnLock    __SpinLock(&__reduce_lock)

/*********************** do's ******************************
  Loop index variables should be declared as register and
  local to the procedure (process).
*************************************************************/

#define cyc_macros(i, a, b, pid, P) { int _start = ((a)/(P))*(P) + pid;  \
                          if (_start <(a)) _start += (P); \
                          for (i= _start; i<(b); i += (P)){

#define blk_macros(i, a, b, pid, P) { register _start, _end; \
                _end = (b- (a))/(P) + ((b- (a))% (P)>0); \
                _start = a + _end*(pid) ; \
                _end = min(a+ _end*(pid+1), b);\
                for (i = _start; i<_end; i++) {

#define blk_cyc_macros(i, a, b, B, pid, P) { for (i=a, i<b; i++) \
                if (pid == (i/(B) % (P))) {

#define do_cyc(i, a, b) cyc_macros(i, a, b, myPid, PROCS)

#define do_blk(i, a, b) blk_macros(i, a, b, myPid, PROCS)
```

```
#define do_blk_cyc(i, a, b, blkSize) blk_cyc_macros(i, a, b, blkSize, myPid, PROCS)

#define parallel(i) { i = myPid; {

/**************************** ends **************************/

#define enddo }; __FLUSH_ALL; __BARRIER;};
#define endby (data, size)  }; FlushCache(data, size); __BARRIER;};

#define seqon(pid) {if  (myPid == (pid)){
#define seq seqon(0)


#define __WAIT__ { register int __WaitTmp1, __WaitTmp2, __WaitI; \
                __WaitTmp2 = 0; \
               for (__WaitI = 0; __WaitI <4; __WaitI ++) \
                       __WaitTmp1 = ( __WaitTmp2++) % 1001; }

#define xlock(lock)     XMEM(1, lock)
#define xunlock(lock)   XMEM(0, lock)

/************************ Fine-grain synchronization ********************/


#define WAIT(pid, p) SynWait(&(__locks[pid]), &(__p[pid]), p)
#define SET(p) SynSet(&(__locks[myPid]), &(__p[myPid]), p)

#define RESETP {__BARRIER; __XMEM(0, &(__p[myPid])); __BARRIER;}
#define ResetCounters {__BARRIER; __XMEM(0, &(__p[myPid])); __BARRIER;}

/****************************** debug *********************************/
#define Where(s) {printf("%s on proc %d \n", s, myPid); fflush(stdout);}

/*******************************************************************
 *                  distribution.m:                                *
 *  M4 macros for block and cyclic row distributions for 2-dimensional *
 *  arrays                                                         *
 *******************************************************************/
define(dist_1D,
    `ifelse($2, cyclic,
    `#define $1($3, _j) $10[(($4)/PROCS)*(($3)% PROCS) + ($3)/PROCS][_j]',
        `ifelse($2, block,
           `#define $1($3, _j) $10[$3][_j]')')')

/*******************************************************************
 *                      main.c                                      *
 *                                                                  *
 *******************************************************************/
#include        "numacros.h"
SharedVars;
extern void UserMain();

int options(argc,argv)
int argc;
char *argv[];
{
  int i, tmp;
  for (i = 1; (i < argc) && ((argv[i][0] != '-') || (argv[i][1] != '-')); i++);
```

```
   __numprocs = MaxProcs;
  if (i > argc)        return argc;
  for (i = 1; (i < argc) && (argv[i][0] == '-'); i++) {
      switch (argv[i][1]) {
        case 'p' :  __numprocs = atoi(&argv[i][2]);     break;
        case 'P' :  __pages = atoi(&argv[i][2]);        break;
        default :  fprintf(stderr,"skipping unknown switch '%s'\n",argv[i]);
                   exit(1);
      }
  }
}

main(argc,argv)
int argc;
char *argv[];
{ int i;
  SystemCode_type rc;

  __pages = Pages;
  options(argc,argv);

  InitBarrier(&__bar, __numprocs);
  InitBarrier(&__reduce_bar, __numprocs);
  InitSpinLock(&__lock_var);
  for(i=0; i < __numprocs; i++) InitSpinLock(&(__locks[i]));

#ifdef HURRI
  for(i=1; i < __numprocs; i++)
      StartProcess(UserMain, MIN_REGION_SIZE, USER_PRIORITY, 3, argc, argv, i);
  UserMain(argc, argv, 0);
#else
  for(i=0; i < __numprocs; i++)
      UserMain(argc, argv, i);
#endif
}
```

Appendix B

Parallel Quick Sorting by Overpartitioning

This appendix gives the source code of PQOP written in KSR C. It is written in a readable fashion without hand tuning for specific compilers and the KSR1 machine-dependent features. Porting this code to other scalable shared-memory multiprocessors only requires some changes to the synchronization variables (in the machine dependent part) and to the thread creation (in the `main` function).

```
#include <stdio.h>
#include <unistd.h>
extern char *optarg;
extern int optind;
extern int optopt;

/********************** machine dependent **********************/
#include <pthread.h>
#include <ksr/pmon.h>

#define MAXPROCS        64
#define PAGE_SIZE 4096 /* integers */

__align128 __shared pthread_mutex_t lock;
__align128 __shared pthread_barrier_t barrier;
__align128 int selfI = 0;

#define BarrierIn       pthread_barrier_checkin(&barrier, myPid)
#define BarrierOut      pthread_barrier_checkout(&barrier, myPid)
#define Barrier         { BarrierOut; BarrierIn;}
#define Lock            pthread_mutex_lock(&lock)
#define UnLock          pthread_mutex_unlock(&lock)
#define HLock(var)      _gspwt(var)
#define HUnLock(var)    _rsp(var)
#define timer() (all_timer()/(20000) /* mini seconds */)

/************** some macros for ease of programming ************/
#define min(a, b)       (((a)<(b))?(a):(b))

/* self scheduling macros. "selfI" sould be set to 0 before using */
#define SELF(i, a, b)\
          while(1) { HLock(&selfI); \
            if (selfI >=b) { HUnLock(&selfI);  break;} \
            i = selfI; selfI++; HUnLock(&selfI); {
```

```
/* partitioning iterations [a--b] among P processors */
#define BLK_PARTITION(a, b, s, e) \
          { int _size = (b-(a))/(P) + (((n)%(P))>0); \
            s = a + _size * myPid; e =   min( a + _size*(myPid+1), b);}

/* do_blk parallel loop */
#define DO_BLK(i, a, b) \
          { int _s, _e; BLK_PARTITION(a, b, _s, _e); \
            for (i = _s; i< _e; i++){

#define SEQ     { Barrier; if (myPid == 0) {
#define END }; Barrier;}

/******************* global variables for PQOP ******************/
int * A, *B;
int n = 1024*1024;      /* total number of integers to be sorted */
int s = 3;              /* oversampling ratio                    */
int k = 4;              /* overpartitioning ratio                */
int P = 1;              /* number of processors                  */

int * pivots;
int * loc[MAXPROCS+1], * l[MAXPROCS+1];
int * Loc, * L, * L_permut;

#define NUM_PARTITIONS (k*P)
#define NUM_SAMPLES (s*k*P)

#define Swap(a,b) { tmp = a; a = b; b = tmp; }

/* sequential quicksort */
void
quicksort(int A[], int start, int end)
{   int lo, hi, tmp, pivot;
    if ((end - start)>1) {
        lo = start;
        hi = end;
        pivot = A[start];

        while (lo < hi) {
            while ((lo < hi) && (A[lo] <= pivot)) lo++;
            while ((lo < hi) && (A[hi] >= pivot)) hi--;
            if (lo < hi) Swap(A[lo], A[hi])
            else if (pivot< A[lo]) Swap(A[start], A[lo-1])
            else Swap(A[start], A[lo])
        }

        quicksort(A, start, lo-1);
        quicksort(A, lo, end);
    } else if ( ((end-start) == 1) && (A[start] > A[end]))
        Swap(A[start], A[end])
}

/* The quicksort_permut function is used to build a task queue
   in decreasing order of sublist size. It is similar to quicksort,
   except that the permutation is kept in the perm array
*/
void
```

```
quicksort_permut(int A[], int start, int end, int perm[])
{   int lo, hi, tmp, pivot;

    if ((end - start)>1) {
        lo = start;
        hi = end;
        pivot = A[start];

        while (lo < hi) {
            while ((lo < hi) && (A[lo] >= pivot)) lo++;
            while ((lo < hi) && (A[hi] <= pivot)) hi--;
            if (lo < hi) {
                Swap(A[lo], A[hi]);
                Swap(perm[lo], perm[hi]);
            } else if (pivot> A[lo]) {
                Swap(A[start], A[lo-1]);
                Swap(perm[start], perm[lo-1]);
            } else {
                Swap(A[start], A[lo]);
                Swap(perm[start], perm[lo]);
            }
        }

        quicksort_permut(A, start, lo-1, perm);
        quicksort_permut(A, lo, end, perm);
    } else if ( ((end-start) == 1) && (A[start] < A[end])) {
        Swap(A[start], A[end]);
        Swap(per[start], perm[end]);
    }
}

/*******************************************************************
   PQOP Step 3: Partitioning.
        The sorted pivots are used to partition array A[start..end].
        The splitting positions are returned in the loc array,
        so that (pivots[i] >=  A[loc[i-1]+1 .. loc[i]]) and
           (pivots[i] <=  A[loc[i]+1   .. loc[i+1]])
*******************************************************************/
void
quick_part(int A[], int start, int end,
          int a, int b, int pivots[],
          int loc[],
          int bound)
{   int lo, hi, half, tmp, pivot;

    if ( end> start) {
        lo = start;
        hi = end;
        half = (a+b)>>1;
        pivot = pivots[half];

        while (lo < hi) {
            while ((lo < hi) && (A[lo] <= pivot)) lo++;
            while ((lo < hi) && (A[hi] >= pivot)) hi--;
            if (lo < hi) Swap(A[lo], A[hi]);
        }

        if (A[lo] < pivot) lo = min(lo+1, bound);
```

```
        loc[half] = lo;
        if (b>a) {
            if ((b-a) >1)
                quick_part(A, start, lo-1, a, half - 1, pivots, loc, bound);
            quick_part(A, lo, end,  half + 1, b, pivots, loc, bound);
        }
    } else
        for (a; a<=b; a++) {
            if (A[start] < pivots[a])
                 loc[a] = min(start + 1, bound);
            else loc[a]= start;
        }
}

/*************** random number generator *******************/
int para_rand(int * z1)
{    long int temp;
#define a_rand  314159269
#define c_rand  453806245
#define bits31  0x7FFFFFFF
     temp = (*z1) * a_rand + c_rand;
     return( (int) ( temp & bits31));
}

void
print_array(char * name, int A[], int a, int b)
{   int i;
    for (i=a; i<=b; i++)
        printf("%s[%d] = %d\n", name, i, A[i]);
}

int check_order(char *name, int C[], int a, int b)
{    int i, err = 0;

     for (i=a; i<b; i++)
#ifdef DEBUG
          if (C[i]>C[i+1]) {
              printf("%s[%d]  = %d  err  \n", name, i, C[i]);
              err ++;
          } else printf("%s[%d]  = %d  \n", name, i, C[i]);
#else
          if (C[i]>C[i+1]) err ++;
#endif
     return (err);
}

void
PQOP(int A[], int B[],
     int n, int s, int k, int P,
     int pivots[],
     int * loc[], int * l[],
     int Loc[], int L[], int L_permut[],
     int myPid)
{    int partitions = NUM_PARTITIONS;
     int samples = NUM_SAMPLES;
     int i,j;
     int start, end;
     int seeds = myPid | all_timer();
```

```
     int err;
     long int t[10];
#define start_timer 0
#define end_timer   1

     /* initialize array and touch array B on each page */
     DO_BLK(i, 0, n)
     seeds = A[i] = para_rand(&seeds);
        if ((i % PAGE_SIZE) == 0)
            B[i] = i;
     END

     t[start_timer] = timer();

     /* one processor selects pivots */
     SEQ
         int sample_step = n/samples;
         for (i= sample_step/2, j=0; i<n; i+= sample_step, j++)
             pivots[j] = A[i];
         quicksort(pivots, 0, samples -1);
         if (s >1)
             for (i=0; i<partitions; i++)
                  pivots[i] = pivots[i*s + s/2];
     END

     /* use the pivots to do partitioning */
     BLK_PARTITION(0, n, start, end);
     quick_part(A, start, end-1, 0, partitions-2, pivots, loc[myPid], end);
     loc[myPid][partitions] = end;

     for (i = partitions-2; i>=0; i--)
         loc[myPid][i+1] = loc[myPid][i];
     loc[myPid][0] = start;

     for (i = 0; i<partitions; i++)
         l[myPid][i] = loc[myPid][i+1] - loc[myPid][i];

     /* one processor builds a task queue */
     SEQ
         for (i = 0; i<partitions; i++) {
             L_permut[i] = i;
             L[i] = 0;
             for (j = 0; j<P; j++)
                 L[i] += l[j][i];
         }
         Loc[0] = 0;
         for (i = 1; i<partitions; i++)
         Loc[i] = Loc[i-1] + L[i-1];

         quicksort_permut(L, 0, partitions-1, L_permut);
         while (L[partitions -1]==0)  partitions --;
     END


     /* sorting */
     SELF(i, 0, partitions)
         int k1, B_left, b_l,  B_right, b_r, pivot, blk;
```

```
        /* copy a sublist to its destination location. during
           copying, the sublist is also partitioned into two.
           then apply quicksort on each of them.
        */

        blk     = L_permut[i];
        B_left  = b_l = Loc[blk];
        B_right = b_r = Loc[blk] + L[i] -1;
        pivot   = A[loc[myPid][blk]];
        for (j = 0; j<P; j++) {
            for (k1 =loc[j][blk]; k1 <loc[j][blk]+l[j][blk]; k1 ++) {
                if (A[k1] <pivot)
                     B[b_l++] = A[k1];
                else B[b_r--] = A[k1];
            }
        }
        if (b_l > B_left)
            quicksort(B, B_left, b_l-1);
        if (b_r < B_right)
            quicksort(B, b_r+1, B_right);
    END

    t[end_timer] = timer();

    Barrier;
    printf("sorting time %d on %d \n", t[end_timer] - t[start_timer], myPid);

    Barrier;
    if (err = check_order("B", B, start, min(end+1, n-1)))
       printf("total errors found %d on %d\n", err,  myPid);
    Barrier;
}

void *UserMain(int pid)
{
    PQOP(A, B, n, s, k, P, pivots, loc, l, Loc, L, L_permut, pid);
}


int main (int argc, char *argv[])
{   int c, i, maxprocs;
    int * tmpP;
    pthread_t pid;
    processor_name_t ptable[MAXPROCS];

    optarg = NULL;
    while ((c = getopt(argc, argv, "p:P:n:N:k:K:s:S:")) != EOF) {
        switch(c) {
        case 'p':
        case 'P': P = atoi(optarg); break;
        case 'n':
        case 'N': n = atoi(optarg); break;
        case 's':
        case 'S': s = atoi(optarg); break;
        case 'k':
        case 'K': k = atoi(optarg); break;
        default:  printf("Usage: %s %s %s %s \n", argv[0],
                         "-n <# of integers to be sorted>",
```

```
                     "-p <# of procs>",
                     "-s <sampling ratio>",
                     "-k <partitioning ratio>");
   }
}
maxprocs = pthread_procsetinfo(ptable, MAXPROCS);
if (P<1 || P > maxprocs) P = maxprocs;

/* allocate memory for arrays A and B */
if (n< (s*k*P)) {
    printf("parameters s and/or k are too big \n");
    exit(2);
}
if ((tmpP = (int *) malloc(2*(n+32)*sizeof(int))) <=0) {
    perror("allocating arrays A and B");
    exit(1);
}
A = tmpP;
B = & tmpP[n+32];

/* allocate memory for arrays pivots, loc, l, Loc, L, and L_permut */
if ((tmpP = (int *) malloc(sizeof(int)*(
                      NUM_SAMPLES + 32  /* pivots           */
           + 2* P*(NUM_PARTITIONS + 32) /* loc and l        */
           + 3*   (NUM_PARTITIONS + 32) /* Loc, L, L_permut */
     ))) <=0)
{
    perror("allocating pivots and others");
    exit(1);
}

pivots = tmpP;
tmpP   = &tmpP[NUM_SAMPLES + 32];

for (i=0; i<P; i++) {
    loc[i] = &tmpP[ 2*i   *(NUM_PARTITIONS + 32)];
    l[i]   = &tmpP[(2*i+1)*(NUM_PARTITIONS + 32)];
}
tmpP = &tmpP[2*P*(NUM_PARTITIONS +32)];

Loc      = tmpP;
L        = &tmpP[   NUM_PARTITIONS + 32];
L_permut = &tmpP[2*(NUM_PARTITIONS + 32)];

/* initialize synchronization variables */
pthread_barrier_init(&barrier, pthread_barrierattr_default, P);
pthread_mutex_init(&lock, pthread_mutexattr_default);

/* create threads and move them to different processors */
for (i=P-1; i>=1; i--) {
    if (pthread_create(&pid, pthread_attr_default, UserMain, (void *)i)
        != 0) {
        fprintf(stderr,"Error in creating thread %d\n", i);
        exit(-1);
     }
     pthread_move(pid,ptable[i],TH_BOUND_PROCESSOR);
}
pid=pthread_self();
```

```
    pthread_move(pid,ptable[0],TH_BOUND_PROCESSOR);
    UserMain(0);
}
```